RAPID USER INTERFACE DEVELOPMENT WITH THE SCRIPT LANGUAGE GIST

Groningen Dissertations in Linguistics 8

ISSN 0928–0030

Rijks*universiteit* Groningen

Rapid user interface development with the script language Gist

PROEFSCHRIFT

ter verkrijging van het doctoraat in de Letteren
aan de Rijksuniversiteit Groningen op gezag
van de Rector Magnificus Dr S. K. Kuipers
in het openbaar te verdedigen op donderdag
24 juni 1993 des namiddags te 1.15 uur precies
door

Gijsbert Bos

geboren op 10 november 1963 te 's-Gravenhage.

1e Promotor: prof. dr F. Zwarts
2e Promotor: prof. dr J. van den Bos

Preface

As more and more people are using computers routinely, not out of love for the machine, but because they need to, it becomes increasingly important to have easy-to-use, «intuitive» software. Consider, for example, the catalogue of a public library. Most people can find a book if the catalogue is on index cards, but having all titles on-line in a computer has some advantages. For those who need it, it allows looking up books in ways not possible before – and if it is possible, sooner or later someone *will* need it. However, it forces every visitor of the library to use the computer instead of the card-tray.

It's not only card-trays that are being replaced by computers, humans are ousted by them, too. Expert systems are deliberate attempts at making use of the knowledge of a human expert without the need of having him (her[1]) around in person.

Extracting the knowledge from a human expert and putting it in a form suitable for processing by a computer is only part of the problem, however. A computer program only makes sense if the people who would benefit from it are able to use it. Asking someone a question is something we are all very experienced in, but asking a computer a question is a completely different thing.

This doesn't imply that the task of the designer of a «user interface» is to make the computer mimic a human, far from that! Occasionally talking to a computer is indeed the best way of interacting, but more often other ways are much more efficient. The computer can display pictures, charts and other things and the designer of the interface would do well to make use of these capabilities.

That the task of creating a good interface is at least as hard as that of making the computer solve a particular problem is

[1] For ease of reading, only «he» has been used in this text. «He» should be taken to mean «he or she» where appropriate.

proven by the fact that on average about half of the time of writing a program is spent on the interface part (assuming it uses graphics of some kind). Since the interfaces of different programs often have a lot in common, this is an area where specialised tools can make life a lot easier. A number of such tools have been introduced already, most of them meant to be used in conjunction with a particular «window system», such as Microsoft Windows on the PC, or X Windows on UNIX machines. Tools can be targeted at several types of developers. Most often they are meant to be used by professional programmers; these tools usually give access to everything the window system has to offer. Some tools are made for people who only occasionally want to program. These tools offer only limited access to the window system, but they are much easier to use.

One of the most important lessons that every programmer has to learn, is «not to reinvent the wheel». This means that using pieces of software that are available is often better than writing them oneself. This lesson can be extended to the use of tools. Use available tools, preferrably the simplest, since it saves you so much time. Only when you are sure you need more than a particular tool can provide should you switch to another or to doing it yourself.[2]

In spite of the sound advice in the last paragraph, part of this book describes my effort to do just that: reinvent the wheel. Of course, there is a reason and there is a result. The reason is, that to fully understand user interface technology it is best to treat it as children do an alarm clock: take it apart and put it back together again. The result is a perfectly usable tool, with its own advantages and disadvantages.

Although this book talks about human factors and the design of interfaces, it is not about designing interfaces per se, only about implementing them. Designing involves choosing metaphors, creating pictures, choosing colours, selecting keystrokes and making a lay-out for the screen. It is very difficult to do it right and there is no guarantee that you will be able to create nice looking and usable interfaces after reading this book. On the other hand, the tools described in this book are designed to make the task of creating and improving interfaces easy, so you won't lose much time if you have to try a few times before it comes out right.

[2] Another reason for rewriting software could be that the existing software has grown so inefficient from many patches that a complete overhaul saves more than it costs.

An example is the Oberon System, an operating system, window system and programming language that together provide not much more than other systems, but in a much smaller, more consistent and faster package.

Of course, no such argument can apply to user interface development systems.

Acknowledgments

During the time that I did the research for this thesis and while I was writing it I learned a lot more than I was able to put

into the text. I want to thank my colleagues and ex-colleagues of the section *Alfa-informatica*, Jan de Vuyst, Harry Gaylord, George Welling, Peter Blok, Yvonne Vogelenzang, Erik Tjon, Erik Kleyn and Gosse Bouma, for many inspiring talks. I want to thank Gosse especially, for his critical reading of a draft version and for the shower of useful articles and books. Erik Tjon was the first user of Gist, his enthusiastic efforts inspired many small improvements and he also made the first non-trivial interface.

I learned a lot from Kees de Vey Mestdagh, especially about writing articles, going to conferences and the like. I am indebted to the members of *Phonk* for keeping my curiosity alert and offering many other interesting topics.

GRONINGEN, 23 NOVEMBER 1993

Groningen Dissertations in Linguistics (Grodil)

1. Henriëtte de Swart (1991). *Adverbs of quantification: a generalized quantifier approach.*
2. Eric Hoekstra (1991). *Licensing conditions on phrase structure.*
3. Dicky Gilbers (1992). *Phonological networks: a theory of segment representation.*
4. Helen de Hoop (1992). *Case configuration and noun interpretation.*
5. Gosse Bouma (1993). *Nonmonotonicity and categorial unification grammar.*
6. Peter I. Blok (1993). *The interpretation of focus: an epistemic approach to pragmatics.*
7. Roelien Bastiaanse (1993). *Studies in aphasia.*
8. Bert Bos (1993). *Rapid user interface development with the script language Gist.*

To appear in 1993:

9. Wim Kosmeijer. *Barriers and licensing.*
10. Jan-Wouter Zwart. *Dutch syntax: a minimalist approach.*
11. Sietze Looyenga. *Syntax and semantics of nominalizations.*
12. Ale de Boer. *VP-anaphora in contemporary English.*
13. Petra Hendriks. *Comparatives in categorial grammar.*
14. Mark Kas. *The semantics of verbs.*

colofon
Opgemaakt met TEX3.14t en Postscript 2.0,
Illustraties met Xfig 2.1.3
Lay-out: Bert Bos
Lettertypes: Gladiator & Helvetica
Printer: VariTyper 600
Offset-druk: Universiteitsdrukkerij, Groningen
Oplage: 250
23 november 1993

Contents

Chapter 1

Introduction

All user interfaces have to be designed. This seemingly empty statement still holds a lesson, because all too often programmers just implement the first idea they have, without asking themselves if there are alternatives. Programmers are not typical users, and even if they try to place themselves in the user's position, what seems a good choice at first may turn out to be less than optimal in the long run.

User interface design can be approached from different perspectives. There is the view from sociology: how do computers influence the behaviour of their users? Do computers make tasks more enjoyable, easier, and less stressful? Psychology can look at the way people's skills, learning ability and character are cooperating with or working against certain styles of interaction. Ergonomics studies the relation between various measures of people's abilities to perform a task and measurable properties of their tools, both hardware and software. Last, but not least, there is computer science, which concentrates on the hardware and software that makes various styles of interaction possible. Computer science studies efficiency both with respect to the machine and with respect to the user; it creates new methods and develops the coordination between software, documentation and training.

A book on Human Computer Interaction (HCI) could focus on many aspects of the interface between computers and people. This book only treats interfaces (software only) that are created for traditional and widely available hardware, viz., keyboard, screen and mouse. It brings ideas from the above mentioned four disciplines together, but it focusses on the computer science side. Insights borrowed from all of them result in a list of desired features, but it is not possible to give exact rules about what is good in interface design and what isn't: science just has not progressed that far. But we can

create the means to experiment, to create different interfaces easily, so we can empirically find out what is best in concrete situations.

The purpose of the book, then, is to provide means to design good, user-friendly interfaces. An additional goal is to provide these means to programmers and non-programmers alike, in other words: the tools themselves should be easy to use. The book describes various systems that have been created with this or a similar goal in mind and compares them to the system that the author himself developed.

1.1 Sociology

Computers have changed the way people work and continue to change it. They have given rise to new hierarchies among people and new ways to exert power: the designer has power over the user; a local computer expert has power over his colleagues, because of his knowledge. Tasks have been transferred to the computer and different tasks have been given to humans.

Computers provided better communications for some and insulated others. Many of the effects had never been planned and are now being studied as phenomena that have already happened. The results may be used to predict the impact of new machines and new programs. This may lead to recommendations about the role a particular program should assume. Since the role of a program is largely determined by the way it looks on the outside, i.e., the user interface, this research will likely be a source of principles for user interface design.[1]

Conversation Analysis is also a part of sociology. It studies the rules underlying a conversation, both between people and between people and computers. It has obvious implications for natural language interfaces, but many of the principles apply to other types of dialogue as well.[2]

1.2 Psychology

People differ in their ability for learning, for pattern recognition, their tendency to make early judgements, etcetera. Some people like guidance, others like to explore by themselves. These differences and others call for user interfaces that can be tailored for (and by) particular users, although some people may even be scared of the power that is thereby invested in them.[3]

Designers of user interfaces, especially if they come from computer science, are usually not aware of the differences

[1] An interesting effect of E-mail was described by Sproull and Kiesler [1991]. They studied the differences between people working in a team that met face to face and a team that only communicated by E-mail. It turned out that the latter group showed more cohesion, more people took the initiative and there was less fear of showing ignorance and asking for help. Of course, E-mail can also be misused; Simon Been describes some of the dangers to organizations that are unprepared (Been [1993]).

[2] see Wooffitt [1990]

[3] see Browne, Norman and Riches [1990]

among people. For lack of better information, they usually take themselves as examples of typical users. This is fine if they are designing for themselves or fellow-designers, but their view of a system is likely to be very different from the view the user will have.[4] To simplify in a gross way: if those users had the same character as the designer, they would have become designers themselves.

1.3 Ergonomics

Ergonomics or *human factors* is an experimental science, if only because so few theories about the interaction between humans and their tools exist. Experiments are based on certain hypotheses and certain metrics and try to give an objective view of how well, in this case, a program is adapted to its users.

It is very difficult to perform «clean» experiments, because so many factors can influence the perceived performance. For example, the subjects of the experiment may not be motivated, so that no change to the program will have any effect.

Ergonomics is usually associated with hardware: what are the ideal dimensions of office furniture, how large should keyboards be, what is the best viewing angle for a computer monitor (*hard* ergonomics). But it applies to software as well. The choice of colours and colour combinations is an example.[5] Only *soft* – or cognitive – ergonomics is exploited in this text.

Even though it is difficult to come up with hard facts, the experimental *methods* may be of use.

1.4 Audience

This book is intended for researchers and designers of user interfaces, in particular *graphic* user interfaces (GUI's). But the developed system, Gist,[6] is aimed at a wider audience: everyone who wants to create a graphic interface for a new or existing program. To use Gist you neither have to be a programmer nor a designer.

1.5 Preliminaries

The ideas and results presented in subsequent chapters are based on a number of observations. The most important are listed below.

- *The benefits of user interfaces.* A well-designed user interface can bring several benefits to the user. The interface may enable him to do things he never could do before, or couldn't do well. The former is called *enabling*, the latter *augmentation*.

[4] see Hammond, Gardiner et al. [1987]

[5] see De Weert [1988]

[6] Some time ago, the four letters «G I S T» were an abbreviation for something; they are no longer.

- *Software engineering.* The methods used to create a user interface are different from those used to create other types of systems. The formal specification methods usually advocated for software projects are not well suited to user interface design, where experimentation and iterative design is in order.

  Designers and implementors of computer programs typically use a large variety of tools, such as version control software, prototypers, various editors, and document formatters. New tools are most effective if they can be integrated into, or used together with existing ones. Typical software development tools are Make, RCS, WEB, Emacs and integrated CASE tools such as Hewlett Packard's SoftBench.

- *Language influences thinking.* Developing an interface is much easier if there is a good language to do it in. Yacc[7] is a good language for writing parsers, Lex[8] is good for lexical scanners, SGML[9] is good for structuring text. But for (graphic) user interfaces there is not yet such a language. Several (partial) solutions have been put forward, such as HyperTalk,[10] Abstract Interaction Tools,[11] and my own proposal, Gist.

- *Programmers are not designers.* Programmers are demonstrably not the best people to design user interfaces — unless it is for their colleagues — and trained designers are seldom programmers: a clear case where an intermediary is needed, so why not utilize the computer itself?

1.6 Definitions

The user interfaces that are described in this book can be or are implemented on fairly standard hardware. The following is a list of the most important concepts that describe that hardware and the ways in which it is used.

1.6.1 WORKSTATIONS

[7] see Johnson [1975]
[8] see Lesk [1975]
[9] see Goldfarb [1991]
[10] see Apple Computer Inc. [1988], Lasky [1989], Molenaar [1988]
[11] see Van den Bos [1988]

The collection of devices that a user has in front of him is known as a *workstation*. For the purposes of this book we require a workstation to consist of at least a keyboard, a display capable of displaying graphics and a mouse or a mouse replacement.

Graphic displays are usually *raster displays*. A raster display has a rectangular matrix of *pixels,* each individually addressable. A pixel can show 2 or more colours, one at a

Display types — quality

Displays can be classified along many dimensions. Price is one, ergonomic quality is another. To assess the quality, measurements are done and panels of users are asked for their subjective opinions. Quantities that can be measured are — apart from outside things like housing and controls — :

Line width of thinnest lines
The thinner the thinnest displayed line the better.

Relative width of horizontal and vertical lines
They should be the same.

Relative brightness of horizontal and vertical lines
They should be the same.

Relation of width to luminance
When brightness is increased, the lines should not blur.

Jitter, swim and drift
Jitter is the tendency of lines on the screen to quickly move back and forth (within a second); *swim* refers to unsteadiness at the larger time scale of about 10 seconds; *drift* is the amount by which lines move over a period of about 1 minute.

Color alignment (convergence)
The three colours that make up a white line (red, green and blue) should be placed accurately at the same spot on the screen.

Steadiness
The luminosity of a stationary image should not vary from one minute to the next.

Consistent positioning
The position of a line should not change if the image around it changes.

Colour alignment at the edges
The alignment (convergence) of red, green and blue should not be different at the corners and at the center of the screen.

Line width at the edges
The width of a line should be the same when displayed in a corner as when displayed in the center.

Luminosity at the edges
The luminosity of an image should be the same when moved to a corner as when put in the center.

Figure 1.1

time. All images, including text characters, are made from collections of pixels. Even *vector graphics* (images described by mathematical formulae) have to be converted (rasterized) to pixels before being displayed.

In earlier days, there used to be so-called *vector displays*, that could display vector data without rasterization, but they are very rare nowadays. Vector data consists of lines, circles and other simple shapes with very precise positions and sizes, much more precise than any display can show. A raster display must round every part of the figure to the nearest pixel. Of course, the more pixels there are, the more faithful the rendition becomes. Figure 1.3 shows the effect of rasterization.

Display types — capabilities

Displays can also be characterized by features such as size and number of colours. In contrast to the ergonomic qualities of text box 1.1 (page 5), these aspects are normally available to the software and it is possible to have a program act differently based on the type of monitor it displays on. A program can usually get the following information:

Resolution
Which in this context simply means the number of pixels in horizontal and vertical directions. Typical values are 640×480, 1024×768 and 1280×1024.

Physical size of the display
Expressed in millimeters. Not always available.

Aspect ratio
The ratio between horizontal and vertical pixels per inch. If pixels are as wide as they are tall, the aspect ratio is 1.

Total number of colours
The total number of different colours the display is capable of showing, albeit not necessarily at the same time. This varies between 2 and 17 million ($= 2^{24}$).

Number of simultaneous colours
Also called the palette size. The number of colours that can be shown at the same time is often smaller than the total number of available colours. Typical values are 16 and 256.

Figure 1.2

Figure 1.3 The effect of rasterization: the letters «da» in font Courier as they appear on a raster display (enlarged 6×).

Instead of a mouse, other pointing devices can be used, including light-pens, graphic tablets, touch-screens and joysticks. Pointing devices are means for analogue or *continuous* input, a method that is increasingly recognised as essential for many tasks. The keyboard serves for discrete or *symbolic* input. Symbolic input appeals to people's language skills, it has immense expressive power, but requires learning (see also 2.2 on page 19).

Outside computer science, analogue input is often considered old-fashioned. Compare modern radio receivers with digital frequency displays and numeric keys for entering frequencies directly, with older radios with a dial that had to be turned for tuning. The older method is continuous, the new one discrete.

1.6.2 Interaction

Systems that rely only on keyboard input are usually *synchronised,* i.e., they employ a fairly rigid form of dialogue, where the user and the system take turns, under the system's control.

Systems that use a mouse are of necessity *asynchronous,* because the user can move the mouse and click a mouse button at any moment, without waiting for a question from the system.[7] To cope with this, the programs have a central dispatch routine, called an «event loop». Every action by the user – key press, mouse movement and sometimes even other external influences, like a clock tick – is an *event.* All events are queued. They are examined one by one and the program jumps to the appropriate code to handle a particular type of event.

The *user interface* as seen from the programmer's viewpoint is the part of a system that contains the logic to deal with events and the routines that present the output. Exactly where the user interface ends and the rest of the program starts is often difficult to tell. It is comparable to the distinction between syntax and semantics in linguistics. The user interface encompasses the *morphology* and *syntax,* the *semantics* are a matter for the *application program* behind the interface.

To the user, the user interface includes the semantics of the program. To him, the user interface is much more abstract; it is the view that the system presents to him, the way it allows him to do his job. The user interface is an information channel, it could be replaced with an other one. But to the user there is little difference between the messenger and the source of the message.

A *Graphic User Interface* (GUI) is any interface that uses images to convey information. In some contexts, letters can also be regarded as images, so that even if an interface presents only text, it can sometimes still be called a GUI. For example, in many program editors the shape of the letters on screen is purely accidental. If these programs do not use imagery elsewhere they are not graphic; on the other hand, most desktop publishing programs try to display the letters exactly as they will appear on paper. Here the letters can be regarded as images, therefore the program has a graphic interface.[8]

1.6.3 UIMS and UIDE

A User Interface Management System (UIMS) is a system

[7] Much, but not all of the interrupt-driven (or event-driven) nature can be hidden in special libraries and drivers. Usually, programmers rely on these libraries and on external programs to queue the events and transform them into a single stream of commands. Still, the timing of commands can not always be discarded.

[8] Note that the question whether a program has a GUI or not is not determined by the «screen mode» of the computer or the fact that the program is run in a window under a windowing system. The program itself has to make use of graphics.

that helps both in designing and in implementing the morphology and syntax of user interfaces. Sometimes a distinction is made between software that is targeted more at designing and software that is still used when the results are ready. The former would be called a User Interface Development Environment (UIDE). Both can take various forms. Gist, which is developed later in this book, is a language. Some other UIDE's are themselves interactive systems to edit and produce parts of programs (program generators).

A UIMS implements a certain model of interaction, though at a high level. It provides in fact a paradigm of interaction. Interfaces that fit the paradigm can be constructed much more easily with the help of that particular UIMS. The paradigm not only extends to the way a user is supposed to interact with a system, but also strongly to the way a designer or implementer should go about creating the interface.

Most interfaces (and UIDE's) are made by programmers for programmers. The realization that non-programmers could design interfaces and that these people have very different skills from programmers is still very young. There is a definite need for tools for non-programmers – designed by non-programmers – like ergonomists, graphic designers and psychologists: clearly a bootstrapping problem.[9]

For a UIMS to be able to control not only the input but also the output of an application, a way has to be found to interpret and translate the output to graphics. One way is to define *output relations,* for example, between certain values and geometric forms.

Another approach, which was developed for the Gist system, is to treat the output just like the user input, i.e., as *events* to be handled by the interface. Gist limits the output of applications to text. Every line of text is viewed as a message to the interface. Although this requires the application to represent all output as text, as if it were printing to a dumb terminal, it is not, in fact, a big limitation: text output is much easier to program than graphic output and it is flexible enough to allow Gist to create most kinds of graphics in response.

[9] «Bootstrapping» is computer science jargon. It is a solution to the chicken-and-egg problem applied to computers: how to create a program when the program itself is needed to create it. The solution usually involves creating a partial program and using that to create an improved version, then the process is repeated.

1.6.4 Conversation analysis

The interaction between user and computer is a form of dialogue, or conversation. With purely symbolic (read: text based) input, it is usually convenient to describe it with a grammar. Graphic input and output is much more difficult to describe. Still, people expect the interaction to conform to rules, just like a conversation between two people.

Conversation is often based on turn taking. There is a question and an answer, a statement and a reaction. The times at which a turn ends are intuitively felt by the partners. It feels strange when someone misses a cue or interrupts at random times.

In a conversation earlier topics can be referred to by abbreviations, often just a single pronoun. But the rules underlying this are still being discovered. In a user interface the designer has to be careful to ensure that the user and the system have the same idea about what is the current topic.

Feedback is important at different levels in a conversation. The user wants to be sure that the system has *noticed* his action and *understands* it, finally he wants a *reaction*.

1.6.5 Resources

The term *resource* is used in a limited, technical way. It denotes graphical objects like windows and buttons as well as aspects of them, like pieces of text, colours and dimensions. Resources are what GUI's are made of.

A *resource editor,* therefore, is a program that can change some low level aspects of an interface: change the wording of menus, switch colours, sometimes even split a long menu in two shorter ones.

Usually, resource editors are used between invocations of a program. The changes do not take effect until the next time that the program whose resources are changed is started. «ResEdit» on the Macintosh works in this way. Sometimes resources can also be changed interactively. The X Window System includes such an editor, called «editres». The program is pointed at a running program and changes take effect immediately.

1.6.6 Efficiency

There are two types of efficiency at play in a user interface. The first is the ease with which it allows a user to perform a task, *as perceived by the user*. Time plays only a minor role. This type of efficiency can be enhanced by providing sensible defaults, choices adapted to the user's expectations and shortcuts.

Efficiency in a more technical sense is the time required to perform a certain action: how fast can a menu be drawn, how fast can a typed sentence be parsed.

Clearly, efficiency is something to strive for. But there is the strange effect that computers can sometimes be too fast. At least that is sometimes claimed with the following example:

slow compilers force programmers to think more about their code, resulting in fewer edit-compile cycles than programmers who rely on the compiler to find errors, eventually leading to faster development. There is some evidence for this, see chapter 7 of Shneiderman [1987].

1.6.7 Client-server model

Many programs can benefit from the «client-server model». The client and the server are two separate programs that communicate by sending messages, possibly over a network. The server manages some data and does most of the work. It has no user interface. The client usually communicates with the user. The messages between client and server conform to a strict formal syntax, called a «protocol».[10] Usually, there is a choice of clients, so that every user can select the best. The server doesn't care — nor know — which client is used, as long as the protocol is adhered to.

A few examples of client-server combinations are: some distributed relational databases; the X Window System (all X programs are clients of the X server); Usenet news readers based on the NNTP protocol,[11] such as rn and xrn.

The example of the xrn news reader client shows that programs can be clients of more than one server. Xrn uses the NNTP protocol to communicate with an NNTP server and it uses the X protocol to communicate with an X server.

1.6.8 Hypertext/hypermedia

Text is traditionally something to be processed linearly: you start reading at the top and read on until you come to the end. If a book has an index, you can also try to read only those parts that actually interest you. Encyclopedias often indicate in the text of an entry which other entries have related information. If that scheme is combined with a computer, the related entries can be linked by pointers and the «see-also»'s can become commands that look up an entry for you when you point to one of them. The command is then called an (active) *link*.

This is the basis of *hypertext*. The basic idea can be expanded in several ways, e.g., you can have a system that not only allows you to search for information in this associative way, but also lets you add new links, such as notes to yourself about something you found, or explanations for other people.

The next step is to add pictures, graphs, schemas and other graphical material. The pictures can be merely illustrations,

[10] This is a different kind of protocol from the one explained in 4.8.5.2. That protocol is a precise typescript or log of everything that happened during a session between a computer program and a user. It can later be examined to see, e.g., what mistakes were made.

[11] see Horton [1983]

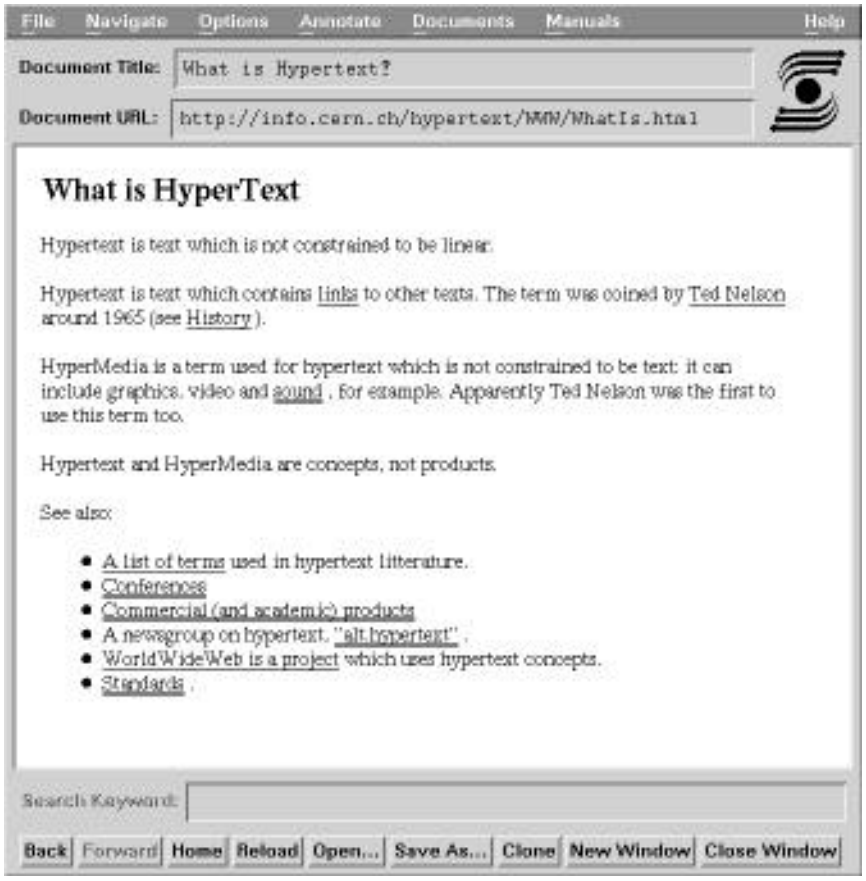


Figure 1.4 *Hypertext in action. This text and the ones in figures 1.6 and 1.5 were retrieved with the World Wide Web hypermedia system, a system that links computers all over the world. WWW information providers install a server; users contact the server with their favourite client program. This screen is from the X Mosaic client. The underlined words are hyperlinks. Double underlines mark links that the user has not yet seen. The «Ted Nelson» link calls up the screen of figure 1.5, the «history» link gives figure 1.6*

but they can also contain links themselves. If the system is expanded even futher to also contain music, speech and animations, it is no longer called hyper*text* but hyper*media*.

The newest hypertext systems have made the links so powerful, that a document can refer to another document that is not even on the same computer. The user doesn't even have to be aware of the fact that activating a certain link causes some

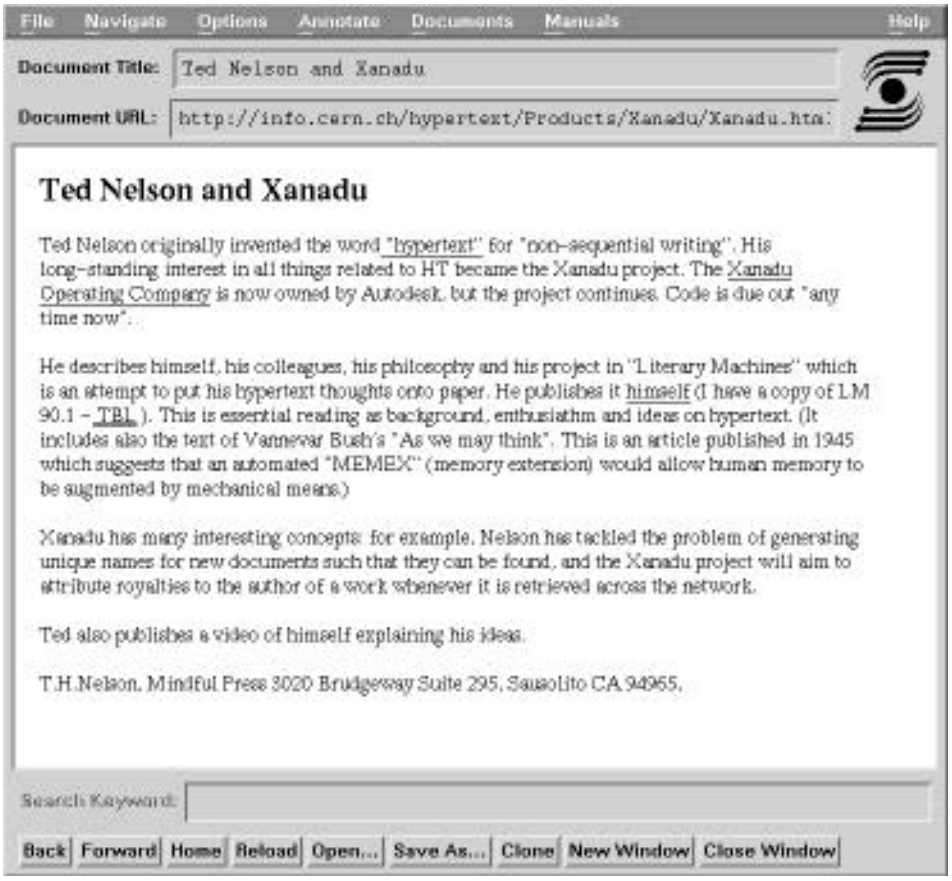


Figure 1.5 The target of the «Ted Nelson» link of figure 1.4

program to fetch the requested information from somewhere else in the world. (See figures 1.4, 1.5 and 1.6.)

Hypertext is thus a poweful concept for structuring information that has complex relations and associations with other pieces of information.

1.7 Overview

Computer hardware has become very powerful and continues to grow in speed and capabilities. There is a definite lack of interfaces that will allow more than just the experts to make use of that power. It is the task of the interface designer to create effective interfaces for different kinds of users. He has few hard and fast rules to go by, but there are some

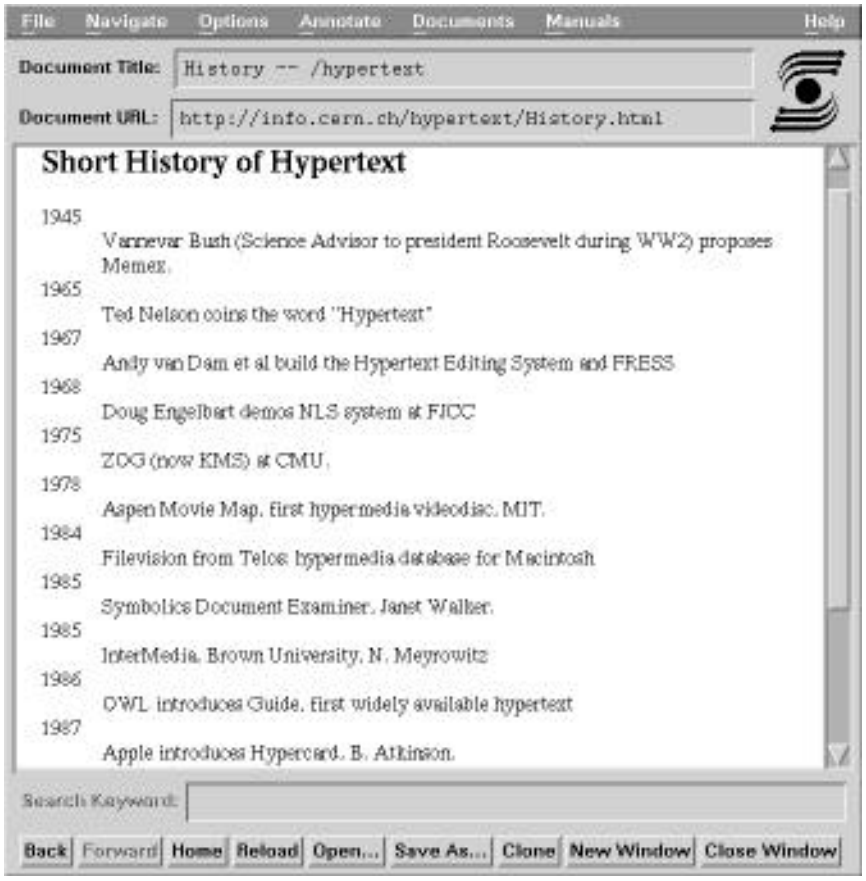


Figure 1.6 *The target of the «history» link of figure 1.4*

rules of thumb. One of the most important technologies is *direct manipulation*, graphics is another, and the combination promises to be even more powerful.

Chapter 3 investigates various tools that have been created specifically to make designing or implementing graphic interfaces easier. Most of them are meant to be used by professional programmers, alongside such tools as parser generators and CASE systems. A few are advertised as being fit for non-programmers, among them the author's proposed system, Gist. Such tools are important, because they allow people that are trained as user interface experts, but who are not programmers themselves, to play an active role in the creation of at least a prototype interface.

Chapter 4 describes Gist and discusses its merits.

Chapter 5 is strictly for programmers. It contains a technical exposé about the way the Gist system is implemented. It is meant for people wanting to improve the system or borrow parts of it. It tries to make clear what have been the considerations behind particular choices of algorithms and data structures and how these choices impose limits on the system's capabilities. There is no claim as to whether this particular implementation is the best possible – it probably is not.

The chapter also describes how a particular installation can be filled (or: configured) with interface elements, since these are not part of the program per se.

For reference, in particular when trying to understand the examples, there is an appendix with the syntax of the script language used by Gist to describe interfaces.

Chapter 2

Human factors & GUI's

More and more people want to – or have to – work with computers and these people naturally want to use all the possibilities the machines offer. In general, they will not be able to follow a course for every program and, moreover, they usually use most features too infrequently to really get experienced. An example are the advisory systems, a new development derived from expert systems. *Expert* systems are meant to be a tool for a limited set of users with knowledge about the field; *advisory* systems are meant for a much larger group of people, seeking information or advice about something they know nothing about. Examples of advisory systems are the city information systems in Maastricht and Tilburg (Netherlands), «Will-writer» (an American system for advice to people wanting to make a will) and programs for computer-aided education.

The developments in telematics also give an increasing number of people access to programs that are located physically far away. Via telephone and television (cable) more and more services are available. Help in the form of human advisors is not offered, however. The information has to be accessible to a very large and heterogeneous group of users. The interface is almost more important than the service that is offered, for people are content with meagre answers, but reject a system in which they get stuck. The program can make no assumptions about the user, except that he possibly never used a computer before and is maybe not even aware that he is using one now. A mail-order company that lets its customers page through the *on-line* catalogue via cable-tv, can send instructions, but is clearly unable to send an instructor to each living room where a user gets into trouble.

The traditional ways of making a computer do what the user wants are through a command language and a menu system.

The drawback of command languages like database query languages (e.g., SQL) and operating system shells (e.g., the Unix shell) is, that they are not appropriate for laymen. Every language has to be learned first and that takes time and, often, guidance. Moreover, one quickly forgets a language when it is not used for some time. Menus solve that problem (the user has to learn and remember little or nothing), but they are useless for complex input, because one quickly gets lost in the multitude of choices.

2.1 The user's perspective

When a program exhibits complex behaviour, the users of the program are likely to build a model that is anthropomorphic. They *project* into the system motives and desires they themselves would have if they were in the system's place. Alternatively, they *transfer* their model of some other well-known person or system to the new system. These are powerful, but often unconscious phenomena.

People have a tendency to see causes and effects in events that occur together or soon after one another, even when the co-occurrence is purely accidental. This may lead to false beliefs that may even extend to things outside the computer, like a belief that a certain program is sensitive to the force with which you press the keys on the keyboard, or even that the program only works if you lift your elbows off the table when it is running. Usually, this points to a lack of trust in the program or an inadequate grasp of its limitations.[1]

The user may fail to see a pattern at all and fall back on describing a system's behaviour as *random*. Most systems other than games aren't random. The *apparent* randomness is caused by gaps in the provided information, by things not discernible to humans (like nanoseconds) or by limitations in human memory: the information that explains a certain behaviour may have been given too long ago, there may have been too much intervening info, or it may have been shown in a position or manner that seemed unrelated to the present situation.

2.1.1 Crammed displays

People's short-term memory can hold very little information, only about six different things at a time. Unless information can be lumped together into *chunks*, overflow is very likely.

Interfaces that require the user to remember too many different things will be almost unusable. Sometimes the user needs to remember things like in what *mode* the program is,

[1] Thimbleby [1990] calls this effect the «magic» model.

how far into the a document he is or in what mode he was before.

One way to improve this situation is to use the computer display as an extension of short term memory. Display editors show how far this can go. An editor like WordPerfect shows just one status line at the bottom of the screen, while e.g., MacWrite shows much more with icons, scrollbars and different fonts.

If the display is to be effective as an aid to the user's memory, the information has to be organized in a way that can be learned and ideally also configured by the user. Not everyone wants the same amount of information. And searching information on the screen shouldn't take too long. Too much information is often called «clutter».

2.1.2 The use of colour

Many people now have colour screens and they expect programs to make use of them, even though many programs do not need colour.

Colour can be used in various ways: to create an aesthetically pleasing display, to create a realistic rendering of an object, to distinguish or identify, or to convey information.

Not all people are the same and they react to colour in different ways. Young people like bright colours; colourblind people have difficulty distinguishing some colours. A few guidelines exist, however. For example, that the human eye is most sensitive to yellowish green, but can only differentiate between a few shades of blue. (See text box 2.1.) A well-designed interface lets users select their own combinations — even though this might result in people painting themselves in a corner, by choosing colours that make essential information unreadable.

A good UIDE should help designers of interfaces by suggesting colour palettes, perhaps by offering sensible defaults.

If the display shows objects and scenes that could exist in real life, it is advantageous to use colours that match those in reality. In publishing or graphic arts it is even essential. Even scenes that clearly cannot exist, can benefit from use of «realistic» colour, for example by shading objects behind other objects. Natural colours induce associations that can help to define the relations between depicted objects.

If the colour doesn't in itself have a clear meaning, it can still be used to distinguish objects or visibly group them together. Repeated objects in a long list or in a series of

Colour and the human eye

Graphics monitors usually treat all colours the same. If they are 24 bit displays, they reserve 8 bits for red, 8 for green and 8 for blue, even though the eye can distinguish greens much better than blues. A few monitiors exist that capitalize on people's low sensitivity for blue to offer fewer levels of blue than of green and red.

The characteristics of the eye lead to a few guidelines for choosing colours: (Macdonald [1991], De Weert [1988])

- Because of the relative insensitivity of the eye for red and blue, one should avoid using these for fine detail.
- Although the eye can see light-dark contrast at a higher resolution, the reliable discrimination of colour is limited to an angle of about $\frac{1}{3}$ degree (about 3 mm at arm's length).
- A luminance ratio of 10 to 1 is optimal for reading. Light characters on a dark background appear thicker than dark characters on a light background.
- Simultaneously focussing on colours of different wavelength is impossible because of imperfections in the lens of the eye. If blue and red (opposite ends of the spectrum) are used together, the red will appear to be in front of the blue. Blue on a red background should therefore be avoided.
- It is hard to focus on edges between colours of the same brightness; seperate such colours with a black line or use colours of different brightness.
- Large patches of saturated colours cause fatigue; it is better to use less saturated colours for large areas. Also, don't put areas of saturated colours next to each other, because they will appear unstable.
- The eye is most sensitive to flicker at the edges of the visual field. To minimize the effect one should use dark colours there, and grays are best.

Figure 2.1

displays are tied together. A single contrasting colour in an environment of subdued tints indicates an important event or object. For this to work, contrasts and bright colours should be applied sparingly.

Red is commonly taken to mean danger. The traffic light analogy is clear. An interface that uses green to indicate errors and red for normal conditions will take some getting used to.

2.1.3 Interesting failures

It is often said that people learn more from mistakes than from successes. Yet in most programs mistakes are not encouraged, because they result in irreversible damage or unhelpful messages.

Still, if a system has a way of showing near-misses (and correcting them), these failures may actually help the user to get a better understanding.

2.2 Classification of interfaces

Two types of interfaces are very promising: natural language interfaces (NLI) and graphic user interfaces (GUI). The idea behind natural language interfaces is that the user can already communicate effectively and without help in his own language; the computer would have to adapt, instead of the user. Graphic interfaces try with the help of images, texts and menus to approximate the (mental) picture the user has of a certain problem. The best known example is probably the *desk top metaphor,* where the screen shows a typical office desk, with files in folders, a note pad, a waste paper basket, etcetera. Often the actions of the user can be reduced to typing a few words and pointing to images and menus with the help of a *mouse* or simply a finger.

Both types of interface could in principle be even further improved by making use of *adaptive* user interfaces. Adaptability may take the form of extensions that allow a user to choose his own style of interaction, or it could be automatic: a «smart interface» that tries to form a model of the user and adapt itself (see also 15).

One way to classify interfaces (or parts of interfaces) is as *atomic, symbolic* or *continuous*.[2] Symbolic interfaces allow building compound commands from simpler ones, according to some abstract syntax. Command languages are an example. Atomic interfaces allow a limited number of commands and nothing more. Menus fall in this category. Continuous interfaces rely on some continuous input device, such as a mouse. Of course, these styles can be combined or emulated in one another.

Symbolic interfaces rely on people's language skills. They therefore require more conscious effort, but are also very versatile. For example, with a system such as the Unix Korn Shell an experienced user can write quite complex commands. The key phrase here is «experienced», since the syntax and semantics of these kinds of interfaces have to be learned and practiced over time. Unless they are natural language interfaces, of course.

Continuous interfaces are much less flexible. There is usually little, if any, provision for building composite actions out of simpler ones. They are nevertheless very appealing, since the actions are quickly learned and can be executed with al-

[2] see Thimbleby [1990]

most no conscious effort – at least in a well-designed interface. The number of basic commands is limited, but a great number of variations that depend on the context can exist. Consider, for example, a drawing program that is operated with a pen (stylus) on a drawing pad. Moving the pen to the left will move something on the screen to the left as well, but depending on the context, the object that is moved may be the end point of a line, a group of images or some control, such as a brightness indicator.

2.3 Goals of user interfaces

Many developers 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.[3] But still, the majority of user interfaces is designed to meet one or more of the following goals, even if often the goals are not made explicit.

- *Easy to use without prior training.* In the most extreme case, someone who knows nothing about the system, but knows enough about the subject domain, could come up to the terminal and start using it.
- *Easy to use after training.* After the user has been taught the principles behind the program (or rather: interface) and has had some practice, he should feel confident with it and have the feeling that it helps, rather than hinders him in doing his work.
- *Easy to learn.* 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.
- *Easy to re-learn after a period of absence.* It is said you never forget how 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.
- *Easy to document.* 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 books, good documentation is especially important. The design of the system could take this into account and avoid letting things happen on the screen that are easier done than described in words.

[3] see Willemse and Lindÿer [1988], p. 48

- *Easy to maintain*. 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.
- *Easy to 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.
- *Fast/minimal effort*. By which is usually meant the number of keystrokes (mouse clicks, mouse travel distance) required for performing some task. Note that this is not the same as the «response time». The response time is a measure of how fast the system can respond to user events and as such it is part of the feedback (see further on). Note also that too few keystrokes can sometimes be disconcerting to the user. In critical situations some redundancy can help prevent errors.
- *Error-preventing*. 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.
- *Flexible*. Some systems are meant to be used by a wide variety of users in many different situations.

Many of these goals are actually contradictory. It is not possible to create a single 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.[4]

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

- *Security*. By limiting user-input and validating it, the interface can help in maintaining the system's integrity.
- *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 elaborate error messages.
- *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.

[4] Unless the user's motivation can be increased enough, of course. It seems that adding a lottery of some kind that occasionally pays out a monetary reward can do a lot to make even difficult systems seem attractive — at least to some people.

Modeless — or «low on modes»?

Modelessness, as described in section 2.3, may not be quite the right word to describe interfaces. All interfaces have, of course, at least one mode, but that is not the real problem. It can be argued that the period between pressing a mouse button and releasing it is a new mode, because everything works differently as long as the button is pressed. But even this can be argued away by noting that this is not a new mode, but rather a suspended state, since *nothing* in the interface works in this situation.

There is, however, a situation where the strive for modelessness contradicts the desireU to make interfaces intuitive. It only applies to naive (i.e., inexperienced) users and it occurs when input is via movement of the mouse over some distance, in particular when moving or resizing objects, selecting from a drop-down menu or performing «drag and drop» operations. In this case the naive user usually prefers to click once to start the action, then move the mouse and finally click the mouse button again to end the action. In the period between the two clicks, the system is definitely in a new mode. Many operations may still be possible and the user can easily be distracted and forget that he is in the middle of an operation.

Experienced users are often more comfortable with the shorter sequence press-move-release. They no longer perceive the operation as a sequence of steps, but as a single action, which is matched by a single motion of the mouse.

Figure 2.2

«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 designer of an interface has many means at his disposal in trying to achieve the goals:

- *Metaphors*. The system tries to mimic something that the user supposedly knows well.
- *On-line help*.
- *Menus* help the user because they list the available options. This is easier, like reading a foreign language is easier than speaking it. Most menus display only text, but the text may be enhanced with graphics or even replaced completely, for example in a menu for selecting colours from a palette.
- *Modeless interfaces*. Which 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.)

- *(Apparent) robustness.* The interface may be able to reassure the user by appearing fool-proof and incapable of doing harm.
- *Consistency.*
- *Conspicuous clues and memory aids.* Colours and icons can help the user to associate particular actions and goals with each other.
- *Simple screen layout.* Desirable if the system must be easy to describe.
- *Resource files* can give people a way to change aspects of the interface without programming and re-compilation.
- *Fixed screens.* This is not the same as simple screens, although it may serve the same purpose. A fixed screen is a screen in which everything has a fixed location.
- *Context-independent keystrokes.* This is a less stringent requirement than modelessness. If there are keystrokes that mean the same from every context, the user can always get back on track if 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.
- *Composite commands* such as macros, can help the experienced user accomplish more with a single command.
- *Clever defaults,* defaults that depend on history, possibly combined with incremental search and auto-completion.[5]
- *Error checking during input* instead of at the end.
- *Feedback.* Examples of feedback are: progress meters, echoing the user's input, error messages and beeps. The type of feedback should be related to the amount of time that has elapsed since the user's action. One of the most important factors for the quality of an interface is the «response time», which is the time between the user ending his action and the first reaction of the system. The shorter the response time the better, even if the system's first response means no more than «yes, I heard you, I'm now working on it.»

[5] see Welling [1992]

- *Configurable interfaces*. Many things can be made customizable: colours, screen layout, fonts, mouse speed, etcetera.

The first decision for the designer of any user interface will be to establish a model of the future user. (Of course, it is easiest if he can take himself as a model.) Will that user be someone with experience in similar programs, will he be trained prior to using the program, will he understand the inner workings of it, etc. A trained user is more interested in speed. To him, a single keystroke is enough to launch a complex operation and a single word of response from the computer can be sufficient. The program does not have to explain what it is doing when it takes some time to complete a calculation. Inexperienced users on the other hand need constant and reassuring feedback.

The next decision is to find a consistent way to map the problem into the world of text and two-dimensional graphics of the computer screen. «Consistent» in this respect means that different parts of the problem that are conceived as similar by the human user, should also look similar on the screen. E.g, if removing a digit is termed «delete» in some menu, then removing a letter somewhere else should not be called «remove».

2.4 Direct Manipulation

Direct Manipulation (DM) interfaces are graphic interfaces with particular goals. Not only is the task presented graphically, but also all commands from the user are given graphically, usually by means of a mouse. The main characteristic of DM interfaces is the fact that they give the user the impression that he can directly manipulate application objects. He is not so much giving commands to the computer, which executes them for him, as he is moving and shaping objects with his own hands. A DM interface should meet other requirements as well. In particular, the goals are the following:[5]

- Learning to use the program should be as easy as possible. Ideally, a demonstration should suffice.
- At the same time, the interface should allow experienced users to proceed very rapidly, maybe by giving them a method for adding their own (composite) functions in the form of macros.
- There should be no need for error messages, either because users can see for themselves what is wrong, or because illegal actions simply cannot be performed.

[5] see Hutchins, Hollan and Norman [1986]

- Everything the user does should have an immediate effect that is enough to inform him whether the action produces the desired result.
- Actions should be reversible, in order to further reduce the user's anxiety. A nice way would be if an action can be done in reverse: e.g., moving an icon to an object induces an action that can be undone by moving the icon back to its previous position.

Although these goals are best approximated by an all-graphics system, it is important to note that, e.g., the well-known spreadsheet also incorporates many of these ideas. On the other hand, not every GUI is a DM interface.

The reasoning behind DM is, that it is easier to use a tool that shows its effect immediately, than to program some action to take place at a later time. On the other hand, if an action takes a lot of tuning, it might be helpful if you can write it down for later reference. A pair of scissors is a handy tool for cropping a photograph to a desired size. But if you have to cut a number of photographs all to the same dimensions, a cutting machine with some knobs to adjust (i.e., «program») is more appropriate.

In a model developed by Donald Norman[6] user activity is modeled as a seven stage process (see figure 2.4):

[6] see Norman and Draper [1986] and Norman [1988]

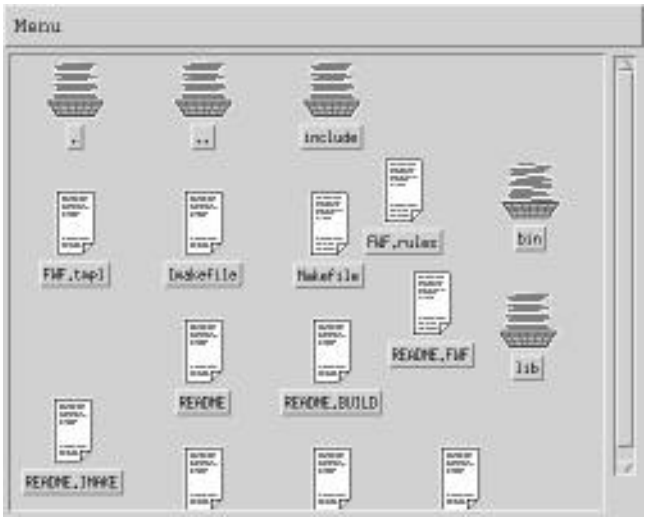


Figure 2.3 A «direct manipulation» file manager shows files as icons that can be moved, copied, etc. with the mouse.

1. forming the goal
2. forming the intention
3. specifying an action
4. executing the action
5. perceiving the state of the world
6. interpreting the state of the world
7. evaluating the outcome

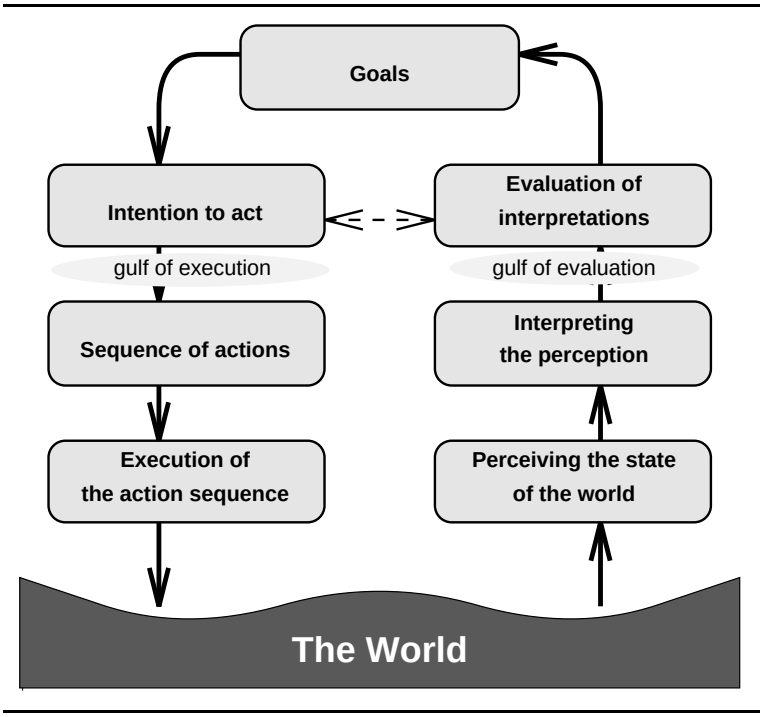


Figure 2.4 The seven stages of user activity, according to the model of Donald Norman

The step from intention to action specification is called the «gulf of execution». It represents the distance between the user's ideas and the commands provided by a system (we assume a computer system, but the theory is more general). The intention must be «mapped» or «matched» to the available actions.

The step from interpretation to evaluation is called the «gulf of evaluation». It represents the translation of the system's messages back to the user's mental model.

In an abstract sense, the goal of DM is to minimize the *mental distance* between *what* a user wants to accomplish

and *how* he actually does it. In other words: the goal is to minimize the effort required to bridge the two gulfs.

One way of bringing the user and the system closer together is to represent (model) the system not as a tool that manipulates an object, but as the object itself. This idea is closely related to *object-oriented programming*. It removes one indirection step. The user should feel that he himself is handling an object, instead of asking the computer to handle it for him. Hutchins, Hollan and Norman [1986] call this *(direct) engagement*.

2.4.1 Virtual Reality

A descendant of DM is *Virtual Reality*. It is currently still too expensive and too complex to be used routinely in interface design.

Virtual Reality (VR) is the name given to a mode of human-computer interaction with as main characteristic that it tries to immerse people in a situation, using imagery that seems to surround them, that makes sound and that changes when they move. The oldest and best known instance of VR is the flight simulator. A flight simulator is a faithful rebuilding of a typical airplane cockpit. It has all the controls and displays found in a real plane, but instead of windows it has large computer screens and the very realistic movement of the simulator is produced by hydraulic lifts mounted underneath the whole system.[7]

A flight simulator is of course a very costly device, though much less costly than real planes. Thanks to the continuing miniaturization of computer hardware and increased processing speed, VR is becoming available for other purposes as well. For example, NASA plans to use it to train astronauts. A related application is to re-create a situation that is actually seen by a robot in a remote or dangerous location. The operator can wear a head-mounted viewing device, consisting of two miniature tv-screens, together with electronic sensors that transmit every movement of the operator to the remote robot.

Other virtual realities are even more «virtual». The computer can simulate situations that can never be real, like moving through space at super-luminal speeds, or looking inside individual molecules. If researchers can manipulate those situations directly, they may gain new insights.

Of course it isn't always useful to put oneself inside a virtual reality so completely; often one wants to alternate between a

[7] see Jacobson [1992], Foley [1987], Hofland [1992]

simulation and an outside view, e.g., to control and record parameters.

It will be some time, however, before an application developer or a user can choose freely between VR and non-VR. Special hardware is required that is currently only partially available and powerful computers are needed that can deal with the complex input from voice, gestures and eye movements and that can animate high-resolution displays in real time.

Still, there are signs that the mouse may get some competition as an input device. Joysticks have found their way into airplane cockpits and trackballs replace mice where desk space is tight. A newspaper article (NRC Handelsblad August 1st, 1991) shows a glimpse of the direction trackballs may be heading: Philips and the Technical University Eindhoven have developed a trackball with force-feedback. When you try to move the cursor in a direction that isn't useful, you feel an increased resistance. The response time of users (on average 175ms with a normal mouse) decreases by about 30%.

2.5 Windows

Windowing is the dividing up of the screen space in smaller, independent areas, that can display different kinds of information simultaneously, without interfering. Sometimes the windows are put next to each other (tiling), but usually windows overlap, causing parts of other windows to be obscured.

Windows can both help the user and slow him down (if there are too many windows that demand his attention). But usually windows are considered an advantage. There are seven major reasons for this:[8]

- More information can be displayed, possibly from multiple sources.
- Information can be combined in various ways.
- Since windows are nowadays usually managed by a separate window manager, the programmer doesn't have to worry about information in other windows, making for simpler programs.
- Windows can function as information storage, where the user can put away things, while working on something else. When the time comes, he will be reminded of what he put on hold there.
- Temporary windows (pop-up menus, dialog boxes) alert the user to certain conditions or remind him of his options, without disrupting the integrity of the display.

[8] see Marshall, Christie and Gardiner [1987]

- Different windows may provide different views of the *same* task.
- A window provides a *context* for interaction, meaning that commands can differ in their meaning in different windows, without being confusing. Different colours, cursor shapes, etcetera, can help differentiate the contexts.

2.6 GUI's

The «graphic» in «graphic user interface» (GUI) refers to the fact that these interfaces make use of pictures, symbols and graphics on the screen that could not be used if the interface were restricted to text. Possibilities include various kinds of lines, graphs, different typefaces and styles (bold, italic), animation, small and large images, etcetera.

GUI's thus offer a richer environment for visually presenting information on screen. In principle, every pixel on the screen could be given its own color and there are often a million or more pixels on one screen. The abundance needs to be controlled, however, and therefore a set of concepts must be developed that describe screen images at a higher level of abstraction. Windows (see the previous section) form the highest level. They group together everything belonging to a single application or a clear subtask of an application. Windows are *interface elements*. Other interface elements are buttons, icons, menus, etcetera. (See the next section.)

Within an interface logically related elements are grouped together. Windows are used for this, but also within a window there can be several visually distinct groups of objects. There are usually special graphic elements available to visually delineate them, such as drop down menus and various frames.

Not just grouping, but the complete layout within a window is important. Some things are best put next to each other, others are better stacked. Some elements should always have the same size and some elements should always be aligned with each other. For example, the elements of a scrollbar (slider, two arrows) should always be in the right order; confusion would arise if the two arrows were interchanged.

The textual elements of an interface are divided into static elements and messages. Static elements are things like labels and titles on other elements. Messages can be further subdivided into status messages, error messages and on-line help.

Status messages provide feedback to the user about what the program is doing and how far it has proceeded. They do not have to draw attention and most users ignore them

anyway, since they tell exactly what the user already expected. They are provided for occasions when the user gets distracted and needs a clue to get back to where he was. For that reason status messages are best put in a fixed location, but clearly removed from the other elements of the interface. A common place is along the bottom of a window.

Error messages require immediate attention and are therefore often overlaid over the other elements, hiding them, approximately at the spot where the users attention is focussed. Often there are levels of severity: warnings indicate possible problems but the user can ignore them if he is sure of himself; normal errors require corrective action; fatal errors signal a unrecoverable problem from which no escape is possible other than aborting and restarting the whole system.

On-line help is usually much more bulky and requires a window of its own, although there may also be a facility for abbreviated or summary information. The balloon help on the Macintosh is an example: when the user knows in principle how to do something but needs to be reminded of some details he can request a summary that will appear in a small balloon that is visually attached to the element about which it speaks.

All elements have a standard or typical form, that can be modified with regards to «details» — also called properties or attributes. Details include colours, type of frame, typography, shape, animation effects, blinking frequency, etcetera. These details can also include part of the elements behaviour, especially with regard to immediate feedback. For example, when a button is clicked, the button can change its colour or frame. Immediate feedback does not require a round-trip to the application. Of course, the real action of the button is not under the control of the button itself, and may require quite different types of feedback.

Animation effects — sometimes called visual effects — can be as simple as the exchange of two colours in a frame to give an effect as if the object moves in or out of the screen. They can be more elaborate, such as objects that «zoom open» when they first appear or slowly dissolve when they are removed again. An icon can cycle through a small sequence of slightly different pictures to show what action is being performed. For example, the printer icon in Hewlett Packard's VUE interface shows a piece of paper moving through the printer when a document is printed and the trash can on the Macintosh briefly bulges when a document is deleted.

2.7 Interface elements

Although most window systems and GUI's use the same kind of interface elements, they often name the elements differently. This section tries to list the most important elements. Many other elements exist, but usually they provide the same functionality as elements from the following list, and they only look differently. Which is not to say that different looks may not be important!

2.7.1 Windows

A *window* is a rectangle on the screen that encloses (usually) a collection of text and images. A window presents a certain view of some related data, e.g., it can contain an image of a car together with technical data such as price and fuel consumption. When a window is drawn on the screen, it obscures whatever was in that area before. This definition of a window is shared by most popular windowing systems, such as Microsoft Windows, GEM and the Macintosh. Only the X Window System uses a different, much more comprehensive concept of window: every button, icon, menu, on the screen is a window.

2.7.2 Boxes & menus

If windows have a more temporary character they are called *boxes*. Boxes usually have less decoration along the borders than windows. A program displays a box if it needs information from the user without which it can't continue. When a box is on the screen, windows and other object are usually de-activated. This is commonly known as a *modal dialog*, meaning that the user must complete this dialog before doing anything else, as opposed to the modeless dialogs of most windows, in which the user may choose which (part of) a dialog he wants to answer.

Boxes are referred to by various other names, such as *alert box*, *alert* and *dialog box*.

Figure 2.5 *Example of a dialog box, this one is in Motif style*

An example where boxes might be used is the following: consider a program that lets a user store results in a file. When the user chooses to save something, the program pops up a box, prompting him to enter a file name or cancel the operation.

A menu contains a short list of *menu items,* usually commands. Menu items can be in three different states: normal, highlighted and unavailable. There is at most one highlighted item, indicating the command that would be executed if the user exited from the menu at that moment. Unavailable items are shown dimmed (also called *grayed*). They are there only to remind the user of commands that are available in other contexts. Most GUI guidelines recommend leaving temporarily unavailable items in the menu, rather than changing the menus depending on context.

Menus behave somewhat like boxes. Indeed thay are often implemented as boxes containing buttons (see below). They pop up on request and thereby de-activate everything else. Since menus are so common, they are often considered separate objects.

There are many different types of menus. Some are permanently visible, others only appear on request. When a menu appears as a direct result of some action by the user, the menu is said to be *posted* by the program. When choices from a menu are made with the keyboard, there are three common methods:

1. the items are numbered and the user types the number;
2. the items have a mnemomic letter, often the first letter of the item, and the user types that letter; or
3. the user moves a highlight or arrow from item to item with the cursor keys and selects with the Enter key.

Methods 2 and 3 are often combined.

When the mouse can be used for menu selection, other types of menus are possible, especially in menus that are not permanently visible.

- *drop-down menus* appear in a fixed place when the mouse is clicked in a special area immediately above that place; hence the name of «drop-down» menu. Sometimes the menu appears as soon as the mouse pointer enters the area — without the need to press a mouse button — but this is generally dismissed as bad design.[9] When several menus are available the sensitive areas are usually

[9] Rumour has it that GEM, Digital Research's window system for the Atari and the PC, which exhibits this behaviour, was designed this way not from any idea about good user interfaces, but simply to ward off copyright claims from Apple, the makers of the Macintosh window system, who chose a click-and-drag type of menu.

collected into a «menu bar», which is in fact a menu of menus.

- *pop-up menus* appear wherever the mouse happens to be when a mouse button is pressed. The advantage of pop-up over drop-down menus is that the user does not have to move the mouse to access the menu, which can save time especially on large screens where the mouse may have to travel quite a distance to reach the menu bar. The disadvantage is that there is no visual clue that a menu is available at all.
- Drop-down menus can sometimes be made permanent, when the user so chooses. Such menus are called *tear-off* menus. The user can detach them from the menu bar and put them elsewhere on the screen.

The mouse not only allows for different ways of posting menus, the selection from the menu can also be made in different ways. There are two possibilities (see also text box 2.2 on page 22): the user releases the mouse button after the menu is posted, moves the pointer to the item he wants and then clicks the mouse again; or he presses the mouse button until the menu appears, than drags the mouse to the correct item while holding down the mouse button and finally releases the button when the mouse is on the wanted item (click-and-drag).

Sometimes the term «pull-down menu» is used for drop-down menu that only supports the latter click-and-drag method.

exit

entrance

Some icons from everyday life and from the computer screen.

2.7.3 Icons

Small pictures called *icons* can be used to remind the user of certain actions or objects. The examples in the margins of this and following pages show a few familiar icons from outside the realm of computers. Usually, there is an action associated with each icon, which is executed when the user clicks the mouse.

Microsoft[9] reserves the name *icon* for pictures that represent objects and uses the name *symbol* for pictures representing actions, although in practice the difference isn't always that clear.

Designing icons is a form of art. It should show something characteristic about the thing or actions, with as little ambiguity as possible and yet be simple enough to be recognized at a glance. The margin on page 34 shows the familiar heart icon. There is nothing about it that connects it to the concept

[9] see Microsoft Corporation [1991]

of love, yet that is what it is commonly taken to mean. On the computer screen the image often has to be formed from as little as 32×32 or 64×64 pixels. To enhance the intelligibility one or two words are sometimes added below the icon itself.

There is argument as to whether the addition of text defeats the whole purpose of icons. I do not believe it does. The purpose of icons is threefold: (1) visually mark a spot in such a way that it is easier to find than when it is only marked by text; (2) give some clue as to the purpose of the represented action or object; and (3) make the interface language-independent.[10] Only (3) is diminished by adding one or two words below the picture. In fact, people that use applications with many unlabeled icons often report that they do not understand the icons and cannot learn them because there are too many. I believe that the much touted «tool bar» that is currently *en vogue* in MS Windows programs is more a result of the strive for putting as many features as possible on the screen. There is barely room for the tiny icons, let alone for text below them.

Some icons are common enough that they can be standardized. Standardized icons are common to all applications on a given computer system. They are part of the – in this case: pictorial – language that a 'computer-literate' user is presumed to know. They do not need text to explain them, instead, they can be used to explain other things. For example, icons for directories, documents and programs have long been standard on the Macintosh. Text that is put next to them is explained by the icon as being the name of a directory, document or program. Motif and OpenLook are also trying to standardize some icons, among them icons that indicate the severity of errors and warnings.

[10] Many designers disregard the possibility of making interfaces more language-independent with icons, by creating pictures that are actually rebuses or puns. For example, one interface I found has the action «launching an application» represented by a rocket about to be launched. In fact, this type of icon also defeats reason 2: giving a clue about the action's purpose.

2.7.4 Buttons

Buttons are perhaps the most flexible elements of a GUI. Buttons come in many varieties, but a prototypical button is a rectangle containing one or two words of text, called the label. It usually has three modes: inactive, normal and highlighted. An inactive button is often shown in gray to indicate that it doesn't do anything at the moment. When an active (normal) button is activated with the mouse, the button is highlighted to provide some feedback.

Figure 2.6 Two buttons, the left one is Motif style, the right one Open Look.

A button can model a switch or toggle. As such, it toggles its appearance or label at each mouse click. It can also be a pushbutton (command button), causing an action to take place when it is clicked on. Sometimes an action occurs as soon as a mouse button is pressed with the cursor over the button, sometimes the action doesn't start until the mouse button is released again. The latter option is the most convenient, because it allows the user to change his mind after pressing a button and move the mouse out of the button before releasing it. Not so often used is a button that causes something to happen for the duration of the press. A simulated piano keyboard would exhibit this behaviour. A third kind of button is the option button, that cycles through two or more settings with each click. The change of setting normally doesn't immediately cause a command to be executed, but it influences the effect of some other command.

Several variations on this scheme are possible. You can omit the border, putting the button in the midst of other words for an effect much like that of hypertext. You can omit the text and overlay a transparent button over an image, presumably indicating that the button will cause something to happen relating to that part of the picture. You can make several buttons act together in such a way that always exactly one of them is selected (toggled), in reminiscence of radio buttons; an array of buttons acting in this way is therefore sometimes referred to as an array of *radio buttons,* a term introduced by the designers of the Macintosh interface.

2.7.5 Images

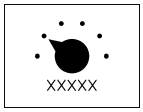
«A picture is worth a thousand words...» — moreover, it is language independent.[11] There are several reasons why pictures might be used in user interfaces, even apart from the possibility that displaying pictures may be the *purpose* of some program.

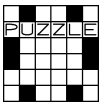
A few nice pictures may help overcome a user's fear when confronted with a new program. The pictures can be used as a lure or as a reward. In the former case they would be visible before the user performed some action, in the latter case they would be shown afterwards. This use of pictures is normally

[11] Not culture independent, unfortunately. The heart-shaped object

is a well-known symbol for «love», but is it universal? I don't know.








associated with (the interface of) educational software, but it might find application elsewhere, too.

Pictures can also be used as explanations. They can contain larger and more detailed versions of icons, or they can give schematics about the functionality of a program. Other examples are: a picture of a keyboard, showing the functions of various keys (this is a standard part of the Macintosh interface); a picture of the mouse, showing the functions of the buttons; pictures of other devices, with accompanying text.

A third type of picture application is the one where the image serves as background for various buttons and other interface elements. The positions of the buttons on the picture indicate their purpose. Examples are: computer simulations, which often show (a schematic representation of) the simulated machine or process, with various other elements superimposed on it, such as counters, clocks, thermometers, et cetera; the educational game in which children are to pair a word and a picture or vice versa; a hypertext-like system in which it is possible to «zoom in» on details of a scene.

There is no sharp dividing line between icons and images, but in general icons are small and highly stylized, while images are larger and more realistic. Icons represents a single object or action, images usually indicate a context or are even purely decorative.

The two type of images, raster images and vector graphics, would both be useful. Raster images could show photographs, cartoons and other free-hand drawings. Vector graphics are normally produced by CAD (Computer Aided Design) programs and used for schematics, maps, etc. Vector graphics have the benefit that they can be easily scaled or otherwise transformed. They can't be used for realistic looking pictures, unless sophisticated imaging tools and considerable computer power are available.

But it is not an easy decision which image format to use. There is a multitude of formats available, some are very easy to manipulate, others are very compact, some can only deal with black and white, others have unlimited colour capabilities. A format like GIF compresses images quite well, but it is limited to a maximum of 256 colours per image, even though the number of available colours is much larger. It is also widely used and well documented. TIFF is another well-known format, less often used thus far, but it doesn't have the limitations of GIF. Formats without compression are much easier to handle, at least if you have enough disk space. PBM and XPM fall in this category.

2.7.6 Sliders

Sliders in the context of GUI's are pictures of slides or valves that can be slid along a scale between two extremes. They provide a way for analogue input. The best known examples of such sliders are *scroll bars*, which are used to slide a text through a window.

Figure 2.7 An example of a slider (Motif style).

Scroll bars actually show *two* things in one picture. They show which part of a large document is visible in a window: the slider is at the top (or left) of its scale if the top (or left end) of the document is visible. However, the size of the slide itself also indicates how much of the document is visible: if the slide is as long as the scale (so that it won't move) it means that all of the document is visible.

Figure 2.8 An example of a scrollbar (Motif).

A simple slider (not a scroll bar) usually has a single parameter: the relative position of the slide, given as a number in the interval [0,1] or as a percentage. Somewhere else in the program some variable is coupled to this parameter. The variable may represent a temperature in a simulation, the brightness of a colour, the volume of sound, et cetera. There are many instances where the exact digits aren't very helpful and analogue input is much more suitable.

Sliders are also used for output. They can be progress indicators for some process or they can indicate such things as volume, brightness and speed.

2.7.7 Prompts (text fields) & editors

A prompt or text field is a short question or other indication that input is requested, with room next to it where a single line of input can be entered. Often the input consists of a number or a word, but it may be more than that. A program might use a prompt to ask for the name of a file or for a password. The prompt may provide a default answer that the user may

accept or edit. Sometimes it also incorporates a template, for example for a zip code or social security number, so that only certain letters or digits can be entered at certain positions.

A prompt is a very small example of an editor. An editor is used to enter or edit an arbitrary number of lines of text, often a complete file. Editors are used, for example, to enter notes to oneself, messages to other people and computer programs.

2.7.8 Lists

A list is a vertical list of items (words), usually with a scrollbar next to it, from which the user can select one or more items. There are five modes of operation (the terminology is borrowed from Open Software Foundation [1991]):

Single selection
Clicking with the mouse on an element selects that element (indicated by highlighting) and deselects the previously selected element. This type of list has much in common with a menu, but usually lists are longer than menus (see figure 2.9) and they contain objects or values, whereas menus contain mostly commands.

Browse selection
Browse selection works like single selection, but in addition, dragging the mouse (i.e., with a mouse button pressed) over the items selects each item as it is traversed.

Multiple selection
Clicking on an element selects it, without deselecting the previous element. Clicking on an already selected item deselects it.

Range selection
Clicking on an element selects that element and deselects the previously selected one. Dragging the mouse selects all elements that the mouse traverses, while deselecting any previous selection. Often it is also possible to click on the first element of a range and then click another button, or a combination of a key and a button, on the last element of the range to select all intervening elements.

Discontinuous selection
Discontinuous selection is a combination of multiple selection and range selection. Clicking on an item adds it the set of selected items, clicking on an already selected element deselects it. Dragging the mouse selects all traversed elements. Dragging the mouse over already selected items deselects them.

Figure 2.9 An example of a list (Motif).

2.7.9 File selectors

A file selector is a special kind of list, that displays filenames from a certain directory instead of a fixed set of items. File selectors are often combined with a prompt and some buttons to create a file selector box.

File selectors are very common in applications, but they are used in different ways. We must differentiate between a number of cases, that may lead to different dialogues. The first case occurs when a program needs the name of a single, existing file; the file will probably be used for input. In the second case, a single file is asked for, which may or may not already exist. Usually, the file is going to be opened for writing and if the file will thereby be overwritten, the program warns about that. Thirdly, the program may ask for a number of files in one action. Nearly always, this means existing files. Finally, the program may ask for a particular set of files, either all files in a certain directory or all file of the same type.

The baseline method

The most straightforward way of selecting a file is simply by typing its name. The program would display a prompt with a blank field, where the user can type the name. This isn't even a particularly bad method: in many cases, the user may already know the name and – especially if the name is fairly short – typing it may be faster and require less effort than any of the methods outlined below.

Improvement 1: defaults

Like in many other situations where a value has to be entered, the file name prompt may offer a default value. The default may be the file that the user has chosen in a similar situation earlier in the program, or the file that is by convention used

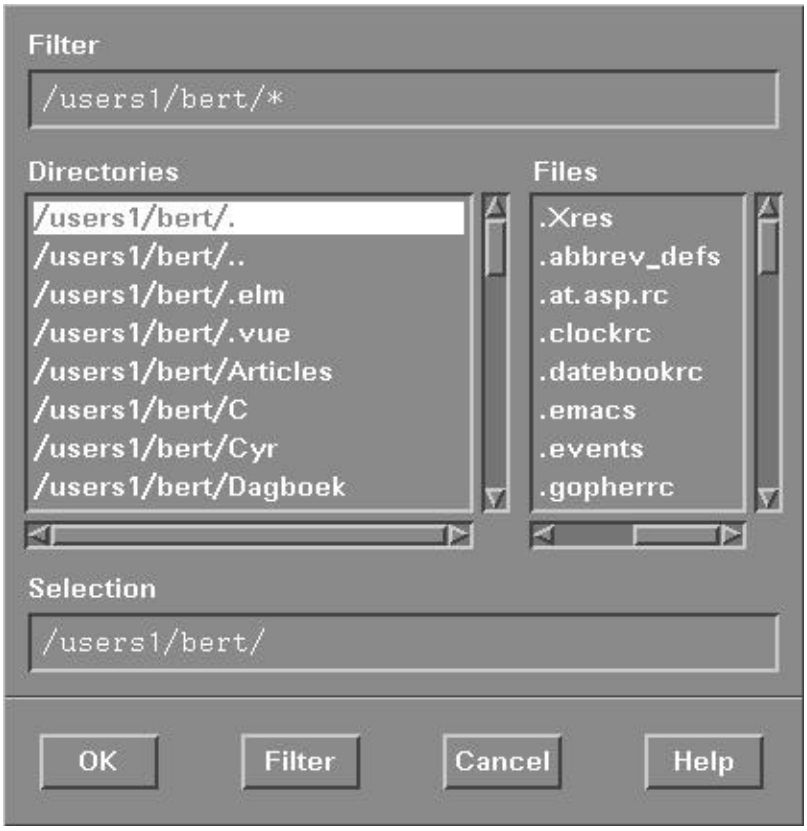


Figure 2.10 *The Motif file selector box. It has two lists, one contains directories, the other files within the selected directory. The prompt at the top can be used to restrict the list to files that match a certain pattern or to jump to a directory that is not in the list. The other prompt shows the currently selected file and can also be used to enter a filename directly. The three buttons at the bottom confirm the selection (removing the box from the screen), apply the filter pattern to the displayed list, and exit the box without making a selection.*

in this situation. For a discussion of intelligent defaults, see Welling [1992]

Improvement 2: auto-completion

The GNU Emacs editor made popular a way of entering file names where pressing the space bar would expand a partially entered name. Typing a unique prefix is thus enough to select a file. Other programs have copied this style of file selection,

such as Hewlett Packard's Softbench (a CASE tool). It is also available in the form of a specialized widget, the FileComp widget by Robert Forsman for the Free Widget Foundation (which is not a foundation at all). The Korn shell (ksh) also provides this functionality, although for obvious reasons it isn't bound to the space bar, but to the escape key.

Improvement 3: selecting from a list

The file selector object in GUI's is a scrolling list of file names, from which the user can select the name he wants. The list can show all files from a directory, all files of a certain type, or a set of files that have been selected earlier. Sorting the list in various ways can help the user to quickly find the file he needs.

Improvement 4: selecting icons

If the list of files is not only used for occasionally selecting files, but becomes the central object of a program, then the files are often laid out two-dimensionally in rows and columns. Every file is furthermore represented by an icon, with different icons indicating different types of files. The most common kind of application that uses this method of presenting files is the (direct manipulation) file manager, a program that allows users to copy, rename, move or delete files (see figure 2.11).

2.7.10 ACCELERATORS

Even in an interface that can be fully controlled with a mouse, it is often convenient to be able to use the keyboard for commands as well. Keyboard equivalents for mouse actions are called *accelerators* (Microsoft: *access keys*[12]). The Motif 1.1 style guide[13] only allows accelerators for menu items. Other mouse actions can still be replaced by key presses, but in a more cumbersome, though not illogical, way, using a mechanism called «keyboard traversal».

2.8 Resources

The Macintosh, GEM and X Windows all have the concept of a *resource*. Although the purpose is the same, the form is vastly different. Resources on the Macintosh are encoded descriptions of various objects used in a program's interface. The resources are stored in the same file as the program itself, and they can be edited with a special program, ResEdit. The main purpose of resources is to make a program adaptable to different languages. ResEdit allows you, among other things,

[12] see Microsoft Corporation [1991]
[13] see Open Software Foundation [1991]

Figure 2.11 An example of a file manager, showing how different icons represent different kinds of files. This view is from Hewlett Packard's VUE*, based on Motif 1.1*

to substitute texts in menus, buttons, etc. GEM has similar resources, but stores them in a separate file. Again, you need a special program to change anything. Resources under X Windows are stored in various places, but most often in a file called «.Xdefaults». This file is a collection of resource names and values. The resource files are databases, with records and field names. There are records for different programs or classes of programs and each record has fields, which may have subfields, etc. Each of the fields has a single value, either a string or a number. E.g., the width of the border of the «Cancel» button of the program «prog» would be specified as: `prog*Cancel*borderWidth: 4`

2.9 Configurability, adaptability & intelligent interfaces

2.9.1 Configurability

If people will be using an interface for any length of time, they may want to configure it according to their personal style and preferences. It is usually not a good idea to offer this option to first time users, but there are good reasons to add *configurability* to some interfaces.

People differ. If a single program is able to put on various faces it may appeal to more people. Situations differ also and they may even evolve. A program may last longer if it can be adapted to many contexts. Let's look at some of the variations among people.[14]

- People may have become used to particular types of interfaces, specific key assignments, specific types of menus, etcetera.
- They may or may not be familiar with the tasks a program is carrying out; some may even know how the program works internally.
- People differ in their adroitness with the keyboard and the mouse. Some may have trouble moving the mouse to exact positions, although it seems this can be trained.
- Not everyone is able to think in terms of procedures and algorithms. Many tasks need planning and formulation of subgoals, but real programming is not everybody's forte.
- Some people are scared of making mistakes or damaging the computer. They will be reluctant to try out options that are not explained sufficiently.
- The internal logic or structure of an interface may not be apparent to people with less pattern recognition skills than the program's designer.
- Even if human help is available, people may be unwilling or unable to use it to its potential.
- People may not be motivated to explore an interface.
- Expectations about a program can be both too high and too low.
- Some people have special skills that could be exploited.

2.9.2 Adaptability

Beyond configurable systems we can envisage smart systems, also called *intelligent user interfaces,*[15] that adapt themselves. Maybe by borrowing techniques from Artificial

[14] see Browne, Norman and Riches [1990]

[15] see Sullivan and Tyler [1991]

Intelligence such interfaces could try to match the abilities and character of a user automatically. To distinguish this capability from the one described in the previous section I suggest we call user-initiated changes «configurability», while we call automatic adaptation «adaptability».

Adaptability is only feasible in complex systems that interact with a user in many different ways over an extended period of time. The program and the user must be able to «get to know» each other.

Psychology and Sociology will have to provide ways to measure skills and knowledge. An attempt at defining such metrics is described in Browne, Totterdell and Norman [1990].

2.9.3 Conversation and role playing

The interface between human and computer should follow the rules of normal conversation. Getting the computer to perform a certain task involves entering into a dialogue. But the two partners – user and computer – need not be peers. The role played by the user and the program can be chosen by the interface designer.

The choice is along a scale from, at the one end, all initiative with the user and, at the other end, all initiative with the computer. The former corresponds to a model of the program as a tool or instrument. The latter might be used e.g., in computer aided education, where the computer acts the part of the teacher.

Very complex programs may be better handled with programs that converse with the user on a peer to peer basis, each with its/his own expertise.

Conversation Analysis (CA) is currently a topic of much research.[16] Since the mid sixties, sociologists have studied the everyday conversations between people. Nowadays people often find themselves also talking to computers and other machines. Therefore, many of the results of CA can and should be applied to user interface design, in order to make the dialogue smoother. Indeed, it works the other way round, too: the computer can be used to study conversation itself in a controlled manner – but that is outside the scope of this text.

2.9.4 Small scale intelligence

As said above, really intelligent systems are difficult to create and will require close cooperation between interface and application – something that is usually to be avoided for reasons of maintainability and ease of design. However, there are some exceptions to this general rule. Some knowledge can

[16] see Luff, Gilbert and Frohlich [1990]

be handled on a small scale by individual *intelligent objects*. For example, in a task where the user repeatedly has to enter words from a limited set, the text field itself could try to find a pattern in the sequence of previously entered words and offer an intelligent default value. Auto-completion and thesaurus-based data-entry are also possible.[17]

As an illustration of the possibilities, consider the following data-entry interface:[18] The task is to enter large amounts of data into a database from hand-written historical archives. The people that enter the data are trained in reading the old handwriting, but that is not enough to fully disambiguate the input. Inconsistent spelling, unreadable writing or simple mistakes by the writers pose problems. Many consistency checks and heuristic rules are used to help the typist.

One technique has the double purpose of saving keystrokes and helping the typist to interpret the writing. A text entry field uses incremental search to search through a list of items for a match. Usually after just two or three letters the intended text is found and the typist can stop typing. The suggested expansion is shown in a lower-intensity color, to differentiate between what the user typed and what the program offers.

If the suggested text is not what the user intended, he can ask for a pop-up list of entries to browse. The list can either be a list that is prepared beforehand, or a list that is built-up during a session from all the entered texts.

2.10 The process of designing UI's

How are interfaces normally designed? An idealized overview is shown in figure 2.12. When the design is undertaken conscientiously, quite a number of steps have to be taken and discussed. Of course in many cases an interface is not really designed, but simply copied from a popular application.

Many methods and tools are applied in the different stages of the design, but many of the tools are used for lack of better ones.

When a new project is started, the goals are formulated and prospective users are asked about their wishes. Among the resulting information should be data about how users currently perform their tasks and how they would like to change that. There are no specialized tools that help in this phase. It is merely a question of talking, reading and documenting.

When the designer or the design team starts with the development, the first ideas are usually drawn «on the back of

[17] see Welling [1991], Welling [1992]

[18] See also Welling [1992], though the description here is of a more recently implemented system, not yet published.

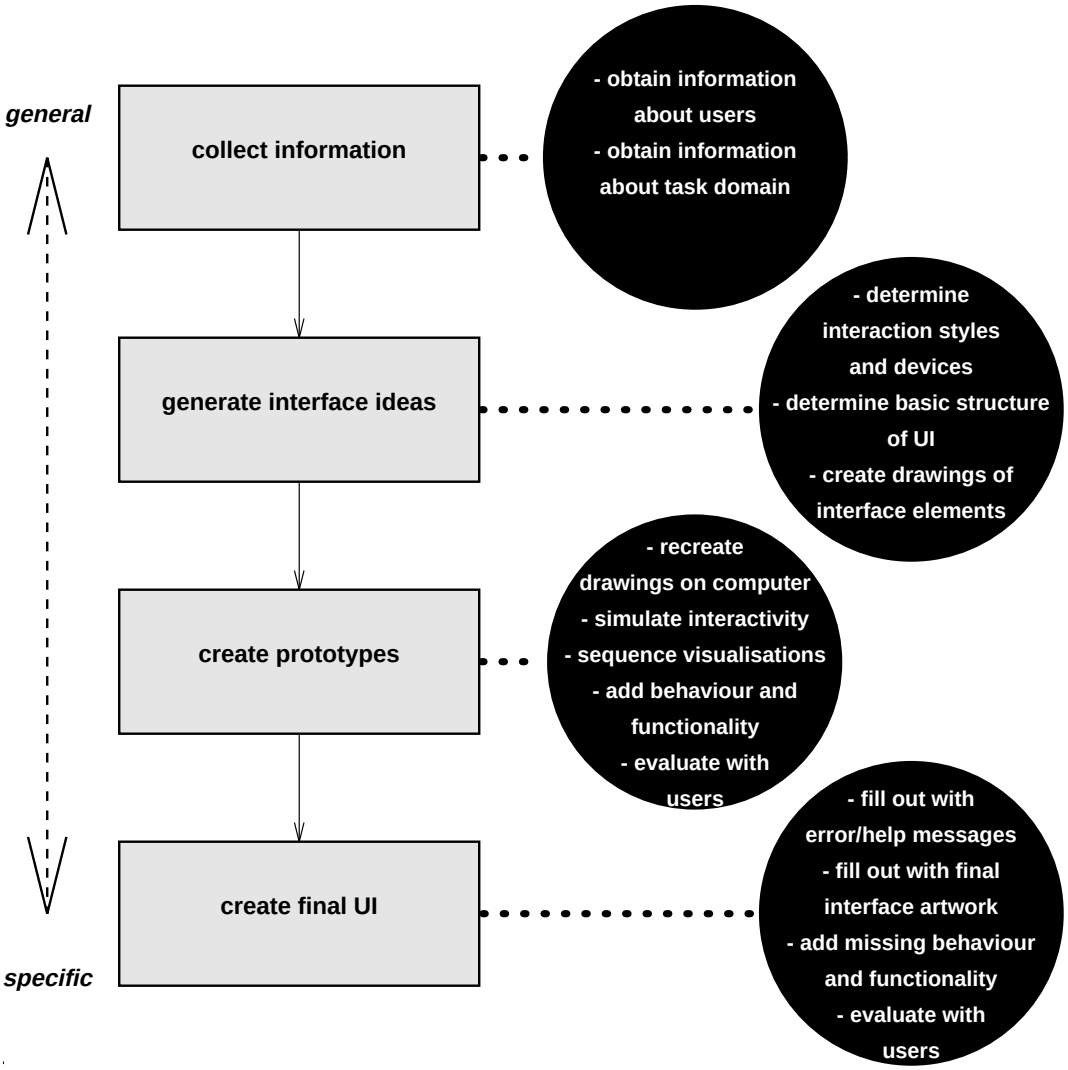


Figure 2.12 A schematic overview of the tasks of an interface designer. The four high-level tasks (boxes) are done more or less in order, although there is a lot of trial and error. The circles give more details. These sub-tasks are not performed in any preset order. (Based on Van der Velden [1992])

an envelope». Only a few of those ideas are written down. The interfaces are represented by — often numerous — sketches.

Interfaces cannot be designed «just right» the first time, so prototypes have to be created, tested and changed several times. Prototypes can sometimes be created with the same

UIDE that will be used for the final interface, but when the designer is not a programmer the UIDE may be too complex for him to use. According to a study by Van der Velden[19] many designers resort to draw/paint programs to draw the various states of the interface. HyperCard (see section 3.3) is also frequently used, at least on the Macintosh.

To help in preventing errors, designers can check with published interface guidelines: books like Brown [1988] that list common do's and don't's. Often there is also a style guide to adhere to, such as Apple's Human interface guidelines[20] or the OSF/Motif style guide.[21]

When a prototype exists that more or less works as desired, the design can be field-tested. People can be observed while they work with the prototype, they can be interviewed afterwards, and they can be asked to «think aloud». If a suitable laboratory is available observation can be done with the help of a one-way mirror or video camera.

If the prototype has been created with the same UIDE that is to be used for the final interface – provided a UIDE is used at all – then the prototype can often simply be enhanced and further filled in. In the other cases a programmer is called in to create the final design in some programming language.

2.11 Style guides and guidelines

One of the goals of HCI research is to come up with a set of guidelines for creating «good» interfaces. So far, published guidelines have been either very vague and general or too specific to a certain context. It is expected that better guidelines eventually will be too subtle or too complex to be easily understandable by designers of interfaces when they are written down. A better method may be to incorporate them into software – either an expert system or a UIMS/UIDS.

The lack of guidelines has prompted a number of hardware and software vendors to create style guides instead. They represent a particular choice of interaction style, that seems to work well in practice. The best known such style guide is probably the one by Apple Computers, called the «Human Interface Guidelines».[22] Other style guides are «Commom User Access» for Presentation Manager by IBM,[23] the «GUI guide» for MS Windows,[24] the OSF/Motif «Style Guide»,[25] the Open Look technical reference[26] and the NeXTStep book.[27]

There is a lot of overlap between the styles for MS Windows, Motif and Presentation Manager, testimony of the fact that they were created in close cooperation and based on a common set of principles (Common User Access, see above).

[19] see Van der Velden [1992]
[20] Apple Computer Inc. [1987]
[21] Open Software Foundation [1991]
[22] see Apple Computer Inc. [1987]
[23] see IBM [1989]
[24] see Microsoft Corporation [1991]
[25] see Open Software Foundation [1991]
[26] see Sun Microsystems [1990]
[27] see Webster [1989]

Chapter 3

User interface development systems

Programs that sport a graphic interface are typically very large and complex, because handling of the bitmapped screen and reacting to all different types of input involves a lot of work. To facilitate the work, sets of algorithms and whole libraries of (graphics & windows) subroutines are being developed (e.g., X Windows) that are often also commercially available (e.g., Microsoft Windows). In all these systems, however, it remains the case that the programmer has to work in a programming language that is primarily geared to working with simple objects like numbers and letters.

Languages that use *object-oriented* techniques are much better suited to handling the complex structures needed for graphic systems. In languages like Smalltalk[1] data and procedures are not considered separate. Instead one starts from *objects* that are defined by the data they can contain as well as the operations performed on them (section 3.1.3).

A different approach is to use a special-purpose language. Such a language is often part of a larger system, a UIMS or UIDE. Some UIDE's even forego the use of a language and let the designer assemble an interface by direct manipulation.

Of course, as soon as interfaces with windows, icons and a mouse become easy to make, the road is clear for new ideas that are waiting already, such as voice-I/O and devices that track the movement of your head or eyes. But even if interface-building tools serve no purpose other than speeding up the development of new ideas, that is enough reason to go ahead with it. Or as Edsger W. Dijkstra said it in his 1972 Turing Award Lecture: «[...] once we have freed ourselves from the circumstantial cumbersomeness, we will find ourselves free to tackle the problems that are now well beyond our programming capacity.»[2]

[1] see Goldberg [1983]

[2] see Dijkstra [1987]

3.1 Software development techniques

Most modern software engineering methods stress the importance of good, precise specification, before actual coding starts. But in the design of user interfaces this technique is not advocated. The method called *iterative design* – meaning design that is regularly adjusted after field tests – is much better here, the main reason being that people are complex and seldom certain about what they want.

3.1.1 Specification vs. iterative design

This presents a dilemma for software developers. How do you reconcile formal specification of one part of a system with the need for experimentation in another part?

The first step towards solving the problem is to be aware of it. Firstly, to take it into account when planning the development effort. Secondly, to be able to trace a problem back to the appropriate part of the design.

This seems to call for a separation of the software in two parts (a modular approach, or even a client-server model, see 1.6.7) and a definition of the interface between them. This solves most of the problems for many programs, in particular those that aim to solve a clear-cut problem. When the task can't be formalised so easily, the problem may require other techniques, because the development of the user interface may still call for changes in what the program does.

Two examples clarify the distinction. First look at a database engine that needs a user interface. The database has been specified precisely, complete with its own algebra. In principle, it can do anything a database needs to do. All that remains for the user interface is to present a particular view. You can choose a radically different interface without it having any consequence for the database.

As an example of a system that can't be specified completely beforehand, consider an electronic messaging system between people in a work group. Experimenting may reveal that users do not just send messages, but that they try to quote and refer to earlier messages, sometimes even several earlier messages. A system that was designed to have two modes of operation – reading mode and writing mode – may need redesign to allow reading and copying of messages even while composing a new one.

A number of techniques are available for assessing the quality of an interface. Depending on available time and resources,

the researcher might use questionaires, interviews, protocols, observation with video, performance testing, etcetera.

3.1.2 Prototyping

The ability to use graphics and a mouse (or any other pointer device) opens up a range of possibilities for interfaces. Should you associate a certain action with an icon, a menu or a button? Should the mouse be «clicked» once or twice? Does moving the mouse have any significance? Do you attach a meaning to the position of the mouse? Do you use colours? Large letters? Animations?

Little can be said in advance about how effective an interface will be. Much depends on the user's experience, the similarity to other programs and, of course, taste. The best way is often just to try and see: make a prototype and test it out, collect protocols of the way people interact with your program, evaluate their reactions and make a new prototype; and so on until you are satisfied.

However, it is not often that you have the time and facilities to actually test a prototype and discard it for something else. And after spending months implementing, who would be willing to do it all over again? The problem, again, is that general purpose programming languages are not very well suited and that even with precompiled libraries of useful routines, the programmer has to do a lot of things over and over again.

3.1.3 Object-oriented programming

In object-oriented programming a program is made up of objects that communicate. Each object represents some data, together with the operations that are defined for that data. The implementation of the operations is hidden inside the object (encapsulation). Each object is of a certain type, called the object's *class*. There may be general and more specific classes. A more specific class is usually a *sub-class* of another class, which means that the subclass *inherits* all operations from the *super-class* and adds some more. It can also add more dimensions to the data that is kept in the object. An operation in an object is activated by sending a *message* to the object.

The same message can mean different things to different classes of objects. This is called *polymorphism*. In contrast, a procedure in a procedural programming language has a unique function. It can only be varied by substituting different values for the parameters of the procedure. E.g., in a graphics program an *icon* can be an object: it contains information about

the appearance of the picture and information about possible changes (different colours, different position on the screen, et cetera).

This method of software design better matches the picture that the user (and the designer!) has of the final interface: a collection of objects on the screen, each of which he can manipulate independently. Smalltalk and the X Toolkit library are examples of object-oriented systems.

3.1.4 Lazy & eager evaluation

There is a difference in the way processing proceeds in the application and interface parts of a program. Most programs are written in *imperative* languages, that support only *eager evaluation*. In this type of program every part of an expression and every argument to a procedure is evaluated, even if later it proves to be unnecessary.[3]

On the other hand, the event-driven processing required by graphic interfaces is more like the *lazy evaluation* type of processing, where computation is postponed until absolutely necessary. This also means that inputs are not processed in the order dictated by the program, but only as soon as and in the order as chosen by the user. The system must be able to work with and display partial results.

3.1.5 Non-deterministic design

An interesting reversal of the role of user interface design is mentioned by Thimbleby [1990] (in box 8.4 on page 166 of his book). *Non-deterministic design* is design for which fitting users will be found afterwards. It isn't that uncommon and it isn't necessarily bad.

It is like an economic market with many producers and many consumers. Many will find a satisfactory partner, because of all the variations.

3.2 Some history

Even in the fifties there were some graphics terminals available, mostly for the display of graphs and other plots. They often had a limited form of graphical input with two thumb wheels that moved a hairline cross on the screen. A more ambitious use of graphics was Ivan Sutherland's Sketchpad program of 1962. It allowed geometric forms to be drawn on the screen, and the corresponding datastructures to be maintained internally.

The use of graphic interfaces got a boost at the end of the seventies with the development of three computers; Xerox

[3] The exception is usually the evaluation of Boolean conditions. In the expression

`if` $A \wedge B$ `then...`

B need not be evaluated when A fails. A and B can be quite complex, even involving user interaction. Lazy evaluation in this case saves the program — and possibly the user — from unnecessarily processing B. Ideally, if A and B both involve actions on the user's part, the user should have the option of answering B before A.

Alto, Lilith and Xerox Star. These computers used a screen that could display high resolution graphics and they received input from an input device that was called a mouse, that was invented at Doug Engelbart's Augmentation Research Center at Stanford, some years earlier. Overlapping windows were also introduced on these machines as was the desktop metaphor. At Xerox Palo Alto Research Center (PARC) the Xerox Alto computer was developed in 1974 for internal use. It was the sort of machine the developers wanted for themselves. It had 64Kb memory – for that time a large and expensive amount – and a black and white graphics screen with about half a million pixels (comparable to modern super-VGA).

The Lilith computer was created in 1977 by a team headed by Niklaus Wirth in Zürich, after Wirth spent a year in the laboratory of Rank Xerox in Palo Alto, where he helped develop the ideas that led to the Xerox Star computer and the first version of the programming language Smalltalk (1976). The Star was the only machine that was sold commercially (in 1981), but it was very expensive and few people could see the benefit of the new system.[4]

The young computer company Apple also tried to turn the Xerox ideas into a commercial product, but their attempt, the Lisa computer, was no success. The value of the ideas was recognized, however, by Alan Kay, one of the major forces behind the Alto, who left Xerox in 1980 and some years later joined Apple to help design the Macintosh computer (1984), the machine that introduced millions of people to GUI's and made them popular. (The hectic years when the Macintosh was born are described in «West of Eden».[5]) In 1985 already other manufacturers followed Apple's example and created machines or software or both for graphic interfaces (Commodore Amiga, Atari ST, Digital Research's GEM for Atari and MS-DOS, Microsoft Windows).

The next notable step in this chronology was again made by Apple. In 1987 the Macintosh was enriched with a piece of software called *HyperCard*. This is a system that allows users to design simple programs that make use of the capabilities of the Macintosh. It is in a sense a replacement for the programming language BASIC, that was often bundled with earlier computers for exactly the same reason.

HyperCard introduced a new metaphor, that of stacks of cards. Each card contains some information and some active elements, such as links to other cards or programs to execute. The user can go forward and backward through the cards or

[4] see Degano and Sandewall [1983] for a description of the Alto.

[5] see Rose [1989]

go to another stack. Predefined elements can be combined with texts and images to create, for example, a database stack or a game stack. When the predefined elements are inadequate, there is a programming language, HyperTalk, with which small programs can be written that are attached to buttons, stacks, icons, etcetera.

HyperCard and HyperTalk are especially important, because they are explicitly meant to be used by end-users, not programmers. The HyperTalk language is kept simple and readable.

The Trillium user interface design environment created at Xerox PARC in 1983 was one of the first systems specifically for use by the designers of interfaces. In this case the goal was to simulate the interfaces of various machines, such as copiers and printers. Fast prototyping was the main requirement. It is an interpreter – the «Trillium machine» – that manages an interface built up from various types of objects or «frames», that react to changes in their environment, such as the press of a mouse button. The actions in Trillium are coupled to so-called *sensors*. Henderson [1986] describes the Trillium system in more detail.

The X Window System is a combination of a window system and a network system. It uses a client-server model. The server manages a display and displays whatever the clients request. There may be any number of clients, running on any number of machines. The clients own one or more windows. To ease programming, X provides a library of routines for common operations, such as opening and closing windows, drawing lines, writing text, etcetera. Even with this «Xlib» library, programming is very complex, and additional, higher level libraries are provided on top of Xlib, such as the X Toolkit.

X Windows was finally distributed in 1987. It had been growing from 1984 through 10 versions at the MIT (version 10 was released in 1986). The introduction of the X Window System meant that now nearly all UNIX workstations could use the same window system, which made them much more popular.

In 1988 the NeXT computer was introduced: a UNIX workstation with built-in support for sound and digital image processing and also an object-oriented tool for designing interfaces. The tool, NeXT Interface Builder, makes it easier for programmers to create GUI's on this computer.

In 1989 a new portable computer was introduced, the GRiDPad, a so-called pen-computer. It is a computer without a

keyboard, but with a stylus with which you can write directly on the screen. The computer can recognize the handwriting. Since then a number of similar computers have appeared, but the technology, especially the handwriting recognition software, is not yet mature enough for widespread use.

HyperCard had a lot of followers, both on the Macintosh and on the IBM PC under Microsoft Windows. A different approach is taken by the program Matrix Layout, which provides its own graphic routines on the PC and uses them to let users create programs by drawing flowcharts on the screen.

The latest contribution from Xerox is the *Information Visualizer,* a way of representing information on screen that uses perspective drawing and animation to pack more data in a small space. The metaphor is no longer the flat desktop with documents and tools scattered all over it, but a collection of *rooms* with walls, doors, and a floor, containing one or more objects. Perspective drawing allows the interface to show more parts of an object (a directory tree, for instance) by making them smaller. To have a closer look at another part of the structure, you rotate it smoothly until the required part comes into view. Zooming in or out is likewise animated. The perspective and the animation give you additional ways of seeing the relations between parts of an object and these clues are presumably processed unconsciously – and therefore very fast – by parts of your visual system.

It is interesting that the Information Visualizer uses a normal keyboard and mouse and no additional hardware. But the fact that the current – experimental – implementation only runs on a Silicon Graphic Iris graphics computer suggests that the system isn't yet ready for the average workstation (or the other way round: that workstations still need to become more powerful).[6]

Meanwhile, the X Window System has spawned many new developments. The X Toolkit defines *widgets,* which are complete, self-contained implementations of interface elements, contained in libraries that can be linked to a program. The widgets allow programming in a more or less object-oriented way. Various vendors have provided sets of widgets to ease and illustrate programming in their preferred style. Examples are the Motif widget set and the Open Look widget set. There are also many public domain widgets and widget sets, such as that of the Free Widget Foundation.

There are also programs marketed by hardware manufacturers to make programming for X on their machines easier (Hewlett Packard's Interface Architect, e.g.). And there

[6] see Clarkson [1991]

are public domain programs like David E. Smyth's Wcl and Richard Hesketh's Dirt, that try to make widget programming more flexible.

3.3 Current systems

A number of UIMS's and UIDE's is currently available, either commercially, free, or for research purposes. An up-to-date list can be found in Heeman [1992]. UIDE's use different approaches, which can be broadly classified as toolkits, interactive GUI builders and script-based systems. (See also figure 3.1.)

A toolkit is nothing more than a library of routines, that can be called from a program. Toolkits may provide drawing routines, windowing routines, and input routines. Examples are the standard X Toolkit, InterViews, CommonView, MacApp and XVT. **InterViews** has been created at Stanford University.[7] It consists of a set of C++ classes. The most notable feature is the way in which screen layout is specified: it uses the boxes-and-glue paradigm of TEX. **CommonView** is commercial product by Glockenspiel. There are versions for Presentation Manager, MS Windows and X, which means that (almost) the same C++ programs can be compiled under all three window systems. **MacApp** by Apple[8] is a library for Object Pascal and C++ on the Macintosh. **XVT**[9] is a library for use with C. Like CommonView, it offers versions on a number of platforms, viz. X11 with Motif, X11 with Open Look, Macintosh, MS Windows, Presentation Manager and text-mode under DOS and Unix.

Interactive GUI builders are programs that let a programmer create an interface by direct manipulation, i.e., by dragging and dropping interface objects, until the screen looks like the desired layout. Unfortunately, this method can only be used for specifying the appearance of the interface, the dynamics must still be programmed, unless the defaults are good enough. Some examples are Dirt, Druid, NeXT Interface Builder and Visual Basic.

Dirt[10] is in the public domain. Dirt runs under X and outputs a resource file that must be read with Wcl. The behaviour of the interface cannot be specified graphically, but must be programmed in the form of callback routines. **Druid**[11] generates C and Motif UIL (User Interface Language). Part of the dynamics can be specified interactively as well. The **NeXT Interface Builder** «IB» outputs Objective C code. Some standardized dynamics can be entered graphically. **Visual Basic**

[7] see Linton, Vlissides and Calder [1989]
[8] see Shmucker [1986]
[9] see Rochkind [1989]
[10] see Hesketh [1992]
[11] see Singh, Kok and Ngan [1990]

	Research	Commercial
Toolkits	ET++ *Interviews PCE *SUIT WINTERP *X Toolkit	*CommonView GUI_Master *MacApp OI (Object Interface) *XVT
Interactive systems	*Dirt *Druid Fabrik *FormsVBT Garnet Ibuild MoDE Peridot Programming by Rehearsal Serpent TAE Plus UIDE *DIGIS	Builder Xsessory DataViews DEC VUIT Devguide ezX Interface Architect LAF Toolkit *NeXT Interface Builder TeleUSE UIMX ViewEdit *Visual Basic XBuild *Matrix Layout *X-Designer XFaceMaker2
Script-based	HyperNeWS *Tooltool *WCL *Motif UIL *Tcl/Tk *Gist	*HyperCard/HyperTalk NewWave Prograph Serius89 IBM dialog manager

Figure 3.1 *A table of user interface development systems. The table is an extended version of table 2 from Heeman [1992]. The products that are marked with an asterisk (*) are mentioned in the text of this chapter. Gist is the author's own system, described further on in this book. The «script» section contains both non-interactive script-based systems and interactive script-based systems. Additional (commercial) systems can be found in Peddie [1992].*

runs under MS Windows. It uses the programming language Basic for programming the dynamics. **X-Designer**[12] also provides a direct manipulation editor for laying out an interface. It generates C, C++ or Motif UIL. The C or C++ functions that it creates are empty stubs, that must be filled out with additional C code. For that reason, X-Designer can be combined with a CASE tool, called CodeCenter. It is targeted at programmers only. It is difficult to change an interface, since

[12] see Bergmans [1993]

regenerating the C stubs causes any code that was put into them to be lost. Changing the layout or the default resources is no problem, however.

Script-based systems read a description of the interface and either generate a program or directly interpret the script. A «script language» is a special purpose programming language, in this case for defining interfaces. Like all programming languages, it can be either declarative or procedural. Procedural languages make use of algorithms, declarative languages define relations and constraints. Procedural languages are not very well suited for non-programmers, but they may give programmers better control. Most script must be typed in first and are then executed, but some systems allow the script to be changed on the fly.

The Motif UIL is an example of an interpreted language, as are Wcl[13] and HyperCard.[14] **Matrix Layout** is a sort of halfway system between a script and a direct manipulation system. It builds a program by assembling a flowchart interactively. The flowchart is then translated into either an executable program or source code for Pascal, Basic or C. The program can also be executed and previewed directly from the editor. It runs under DOS and MS Windows.

A previous version of **FormsVBT**[15] allowed the designer to switch between a graphic direct manipulation editor and a text editor, where a textual representation of the interface could be edited. In the newest version (July 1992)[16] the direct manipulation editor was dropped and only the editor for the Lisp-like language was kept. It is still possible to get a preview of how the interface will look, by pressing a button in the editor. FormsVBT can only be used with Modula-3 (formerly Modula-2+) programs, which must be specifically written for use with one of the VBT libraries. Figure 3.2 shows a sample interface description in FormsVBT.

Tooltool is a UIMS that runs as a separate process in parallel with the application. The application runs as though it reads input from the terminal and writes output to the screen, but Tooltool intercepts the I/O and passes it through a window. It uses a script only for transforming user input, output is simply displayed in a text window. Tooltool has extensive support for expressions, almost as expressive as C. The script syntax is reminiscent of C in other respects, too. The types of interfaces that can be created with it are limited, however,

[13] see Smyth [1991]
[14] see Apple Computer Inc. [1988]
[15] see Avrahami, Brooks and Brown [1989]
[16] see Brown and Meehan [1992]

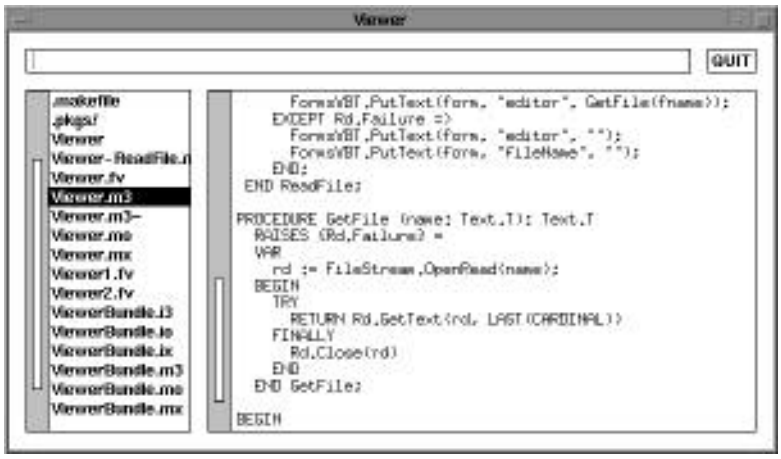


Figure 3.3 The file viewer corresponding to figure 3.2 in use. (This image and the text of figure 3.2 are taken from Brown and Meehan [1992], page 24)

since the number of interface elements is quite small and non-extensible, and since there can be no hierarchy among elements. Tooltool can create a main window and any number of dialog boxes, but all elements in a window or box must are placed directly on the window, they cannot be nested. Tooltool only runs on Suns under Sunview.

3.4 Advantages & disadvantages

Each of the systems mentioned has its own strong and weak points, but they cannot simply be compared. They target dif-

```
(Rim (Pen 10) (ShadowSize -1)
     (BgColor "White") (LightShadow "Black")
     (DarkShadow "Black")
   (VBox
     (HBox
       (Frame Lowered (Typein %fileNameString))
       (Glue 10)
       (Button %exit "QUIT"))
     (Glue 10)
     (HBox
       (Shape (Width 100)
         (Frame Lowered (FileBrowser %fileName)))
       (Glue 10)
       (Shape (Height 200 + inf) (Width 300 + inf)
         (Frame Lowered (TextEdit ReadOnly %editor))))))
```

Figure 3.2 An interface description in FormsVBT. Note the use of the Lisp-like S-expressions and the boxes-and-glue metaphor from TEX. The S-expression defines the interface to a simple file viewer. Each object in the interface must be bound to a procedure with the help of a library routine in the Modula-3 application program.

ferent types of users and claim different parts of the user interface design process. To make a broad categorization, we can distinguish between systems for programmers, for designers and for end-users.

(Professional) programmers can, of course, work with any of these systems. What differentiates them from the other two groups is that they are familiar with the concept of programming, that they understand about efficiency and that they are the designated people when it comes to forcing the last bit of performance from a system. They view an interface as just another program, defined by its datastructures and algorithms, of which only a projection is visible on screen.

By designers are meant those people whose primary expertise lies in the field of human-computer interaction. They may or may not be programmers, but for the purpose of this argument we assume they are not. Section 2.10 describes how they work. They approach user interfaces from the «outside»: the interface consists of the things that can be distinguished on the screen. The relations among them are usually not described in terms of hidden objects or mechanisms, as programmers would do.

When end-users are mentioned as creators of interfaces, the interfaces involved are usually small, single-purpose, and often temporary – the modern day equivalent of batch files or shell scripts.

End-users can also modify existing interfaces. Support for such *configuration* of an interface by the user can be built into the interface, or it can be provided by the same UIMS that the original designer used.

3.4.1 TOOLKITS

Toolkits are clearly targeted at programmers. Usually they are tied to a specific programming language, such as C, C++, Lisp or Prolog. They should be evaluated with respect to their fitness for use in programming. For this reason, and because the use of toolkits presupposes that the application and the interface are developed together, toolkits will not be dealt with in this text.

In fact, toolkits are often used in the creation of other systems. For example, many of the non-toolkit systems have been implemented with the help of the X Toolkit, including my own system, Gist.

3.4.2 Interactive systems

Interfaces have both visual and dynamic – or behavioural – aspects. Visual aspects include size, location, colour, decorations, etcetera. Dynamic aspects include constraints on what actions are available at what time and what action corresponds to which function of the application. Since interfaces are often designed by trial and error, it is seems logical to allow the interface to be created interactively. Many UIDE's employ some form of interactive design, often just for the visual aspects, but sometimes also to specify part of the behaviour.

Interactive systems usually employ direct manipulation to create at least the visual lay-out of the interface. Interface elements are dragged and dropped to their position on the screen, and they can be resized and static texts can be added to them. The relations among interface elements are much harder to specify with direct manipulation. Geometric relations can sometimes be added with the help of metaphors such as blobs of glue (FromsVBT) or springs (DIGIS).[17]

Various systems have also tried to provide some means of adding dynamic behaviour by direct manipulation. E.g. DIGIS allows a diagram to be drawn to specify in what order the various elements of an interface are activated. Interface elements can be collected into panes and then the same kind of diagrams can be created for the relations among panes, thus allowing hierarchical relations. Another system that tries to replace traditional programming with visual programming, although not strictly direct manipulation, is Matrix Layout. It uses flowcharts to specify procedures. The elements of the flowcharts can be selected from menus and inserted at the right place.

Dirt allows functions («callbacks») to be attached to interface objects. The functions have to be created beforehand and must be explicitly exported by the application. In DIGIS, the application must first be described with an object-oriented «Domain Application Model». The objects in the interface can then be linked to objects in the application model by means of various kinds of links or «signals».

3.4.3 Interactive script-based systems

Script languages have a number of advantages over both general-purpose programming languages and DM systems. Script languages are usually much simpler than general-purpose languages. They are easier to learn and they use

[17] see Van den Bos and Laffra [1990] and De Bruin, Bouwman and Van den Bos [1993]

high-level concepts that are close to the concepts in the task domain. And although the use of such languages seems a step backwards from programming by direct manipulation, this is not necessarily the case. Compared to DM systems, they have the advantages of being printable and easily copied in whole or in part. Also, a well designed language can often better express the actions of an object or the interface as a whole than the geometric relations of which direct manipulation must of necessity employ itself. At the very least programming with such languages is much faster than laboriously moving objects, on all but the smallest interfaces.

When the script can be edited interactively – i.e., with changes taking effect immediately, instead of after restarting the system – or when visual aspects of the interface can be modified interactively, the systems are still called «interactive». HyperTalk and Gist fall in this catagory. The combination of HyperCard with the HyperTalk script language seems especially popular.

With script languages the challenge is always to find a good compromise between the flexibility of a full-blown programming language and the learnability of much simpler systems. The designers of the HyperTalk language have tried to make a language that reads almost like plain English. There are no type declarations, but otherwise they have not tried to hide the control structures. Figure 3.4 shows an example. Chapter 4 describes how the dilemma has been handled in Gist.

Figure 3.4 An example of a HyperCard event handler. This script is attached to a card and asks a password when the card is opened. Note the use of the word «it» to refer to the most recently mentioned variable, in this case the result of «ask password».

```
on openCard
  repeat 3 -- the user gets 3 tries
    ask password "give password"
    if it is 11801 -- whatever is
                   -- the coded password
    then
      pass openCard
      exit openCard
    end if
  end repeat
  beep 6
  answer "Entry denied"
  domenu "Stop HyperCard"
end openCard
```

3.4.4 Non-interactive script-based systems

Some interface builders rely solely on scripts and do not allow interactive editing at all. Examples are Wcl and Motif UIL. Wcl extends the concept of resource as it is already present in the X Window System to include the possibility to specify a widget hierarchy and attach functions (callbacks) to other widget. The application must be specially written for use with Wcl, but the interface can then be modified considerably without changing the application.

Motif UIL has a syntax that is reminiscent of C, but the description is purely static. This is strange, since the similarity suggests that Motif UIL can be used for writing algorithms, but this is not the case. No dynamics can be specified, except for the attachment of callback functions that are defined in the application. The description is compiled and linked with the application, which means that a change in interface requires a recompilation of the entire application.

Tcl/Tk[18] uses the traditional UNIX shell as its model. Interface elements are created with commands that have arguments and options. In contrast to normal commands, the commands that create interface elements do not complete before the next command is started. Instead they leave an object on the screen. Figure 3.5 shows a Tcl script, the result is shown in figure 3.7. For comparison, the same interface is also specified in Gist, see figure 3.8. Clearly, Tcl is targeted at people familiar with imperative programming.

[18] see Ousterhout [1993]

```
# mkPuzzle w
#
# Create a top-level window containing a 15-puzzle game.
#
# Arguments:
#    w -        Name to use for new top-level window.

proc mkPuzzle {{w .p1}} {
    catch {destroy $w}
    toplevel $w
    dpos $w
    wm title $w "15-Puzzle Demonstration"
    wm iconname $w "15-Puzzle"
    message $w.msg -font -Adobe-times-medium-r-normal--*-180* -aspect 300 \
            -text "A 15-puzzle appears below as a collection of buttons.
Click on any of the pieces next to the space, and that piece will slide
over the space.  Continue this until the pieces are arranged in numerical
order from upper-left to lower-right.  Click the \"OK\" button when you've
finished playing."
    set order {3 1 6 2 5 7 15 13 4 11 8 9 14 10 12}
    global xpos ypos
    frame $w.frame -geometry 120x120 -borderwidth 2 -relief sunken \
        -bg Bisque3

    for {set i 0} {$i < 15} {set i [expr $i+1]} {
        set num [lindex $order $i]
        set xpos($num) [expr ($i%4)*.25]
        set ypos($num) [expr ($i/4)*.25]
        button $w.frame.$num -relief raised -text $num \
                -command "puzzle.switch $w $num"
        place $w.frame.$num -relx $xpos($num) -rely $ypos($num) \
            -relwidth .25 -relheight .25
    }
    set xpos(space) .75
    set ypos(space) .75

    button $w.ok -text OK -command "destroy $w"

    pack append $w $w.msg {top fill} $w.frame {top expand padx 10 pady 10} \
            $w.ok {bottom fill}
}
```

(continued)

Figure 3.5 *A Tcl/Tk script for the 15-puzzle (see figure 3.7). Compare this script to the one in figure 3.8. (The text after «message» is in reality all on one line.) The Tcl script is clearly procedural, whereas Gist is declarative. Objects get their attributes via slot & filler (or keyword-value) pairs.*

(Continued from 3.5)

```
# Procedure invoked by buttons in the puzzle to resize the puzzle entries:

proc puzzle.switch {w num} {
    global xpos ypos
    if {(($ypos($num) >= ($ypos(space) - .01))
            && ($ypos($num) <= ($ypos(space) + .01))
            && ($xpos($num) >= ($xpos(space) - .26))
            && ($xpos($num) <= ($xpos(space) + .26)))
            || (($xpos($num) >= ($xpos(space) - .01))
            && ($xpos($num) <= ($xpos(space) + .01))
            && ($ypos($num) >= ($ypos(space) - .26))
            && ($ypos($num) <= ($ypos(space) + .26)))} {
        set tmp $xpos(space)
        set xpos(space) $xpos($num)
        set xpos($num) $tmp
        set tmp $ypos(space)
        set ypos(space) $ypos($num)
        set ypos($num) $tmp
        place $w.frame.$num -relx $xpos($num) -rely $ypos($num)
    }
}
```

Figure 3.6

Figure 3.7 *The 15-puzzle as it appears on screen when the scripts of figures 3.5 or 3.8 are excuted.*

```
#!gist

highlight-thickness 0                    # global default

/set (.*) (.*)/:                         # input from application:
    $1$location $2.                      #  set location of $1 to $2
/create (.*) (.*)/:                      # input from application:
    CLONE btn AS $1                      #  create button $1
    $1$location $2                       #  set location to $2
    $1$label $2                          #  set label to $1
    OPEN $1.                             #  open $1

OBJECT window "15-Puzzle Demonstration"
width 405
height 280

OBJECT label message
font "-Adobe-times-medium-r-normal--*-180*"
alignment left
margin 7
shrink-to-fit yes
label "A 15-puzzle appears below as a collection of\n\
buttons.  Click on any of the pieces next to the space,\n\
and that piece will slide over the space.  Continue\n\
this until the pieces are arranged in numerical order\n\
from upper-left to lower-right.  Click the \"OK\"\n\
button when you've finished playing."    # label doesn't do wrapping...

OBJECT board frame                       # this holds the 15 buttons
location "0.5-60 1.0-150 120 120"
background Bisque3
frame-type sunken
frame-width 2

OBJECT button btn (frame) CLOSED         # prototype for button
MOUSE-CLICK: PRINT SELF$label+" "+SELF$location+"\n".

OBJECT button OK                         # button along the bottom edge
location "0 1.0-25 1.0 25"
label OK
MOUSE-CLICK: HALT.
```

Figure 3.8 *This Gist script produces exactly the same interface as the Tcl script in figure 3.5, except that it doesn't contain the puzzle logic. A simple program like the one in Awk in figure 3.9 better describes the puzzle than it could be done in Gist. The puzzle is started with the command:* `puzzle.g puzzle.awk`.
Normally, the location of objects is a matter for the interface. But in this particular program the locations of the 15 buttons are under the control of the application program. The interface in this case only provides a prototype button.

```
#!/usr/bin/awk -f
# receives messages of the form "label x y w h"
BEGIN {
  order = "3 1 6 2 5 7 15 13 4 11 8 9 14 10 12"
  split(order, ord)
  for (i = 0; i < 15; i++)                          # create buttons
    print "create", ord[i+1], (i%4)*.25, int(i/4)*.25, .25, .25
  freeX = 0.75                                      # position of free square
  freeY = 0.75
}
{
  if ($3 == freeY && $2 + 0.25 == freeX) {          # move right
    freeX = $2
    print "set", $1, $2 + 0.25, $3, $4, $5
  } else if ($3 == freeY && $2 - 0.25 == freeX) {   # move left
    freeX = $2
    print "set", $1, $2 - 0.25, $3, $4, $5
  } else if ($2 == freeX && $3 + 0.25 == freeY) {   # move down
    freeY = $3
    print "set", $1, $2, $3 + 0.25, $4, $5
  } else if ($2 == freeX && $3 - 0.25 == freeY) {   # move up
    freeY = $3
    print "set", $1, $2, $3 - 0.25, $4, $5
  }                                                 # else, can't move
}
```

Figure 3.9 *The Gist script in figure 3.8 doesn't contain the puzzle logic. A simple program like this one in Awk better describes the puzzle. The program starts by generating the locations for the 15 buttons and sending 15* `create...` *messages to the interface.*

Chapter 4

Gist

Gist[1] is a UIDE/UIMS developed by the author. It is script-based and partially interactive. Unlike most systems, but like Wcl and Dirt, it doesn't have a built-in set of interface objects. Instead, it provides a general mechanism for describing and linking interface elements. Any concrete installation must, of course, be combined with a particular set of objects, but the set is fully configurable and any widget can be turned into one or more interface objects. Many aspects of interfaces are delegated to the widgets. For example, Gist cannot direct keypresses to specific windows, thus a specific choice of *keyboard focus policy* must be enforced by using the right kind of widgets.

The purposes of Gist are the following: provide an easy to use interface development system, easy enough to be used by non-programmers; it is not necessary that *all* GUI's can be created, only the most common; make developing applications easier, by not requiring the graphic user interface to be part of the application; make the system practical enough for building one-time interfaces, just as people make shell scripts or batch files for one-time jobs; keep it simple, both simple to learn and simple to port.

The competition for Gist therefore not only consists of UIDE's and UIMS's, but also of script languages such as the UNIX shell itself, REXX, Perl, and Tcl.[2]

A distinction must be made between three different concepts that are all called Gist:

1. Gist as a research project – an abstract concept which includes the goals mentioned above plus a number of ideas about how to reach those goals. But which also contains some unresolved issues.

[1] Gist is in fact part of a larger project, covering both the design of the interface elements (widgets) and their assembly into GUI's. Besides Gist, the project includes a widget generator called «wbuild» and an interactive widget development environment called «Tovenaar». Wbuild is already in operation, Tovenaar is not.

[2] see Ousterhout [1993]

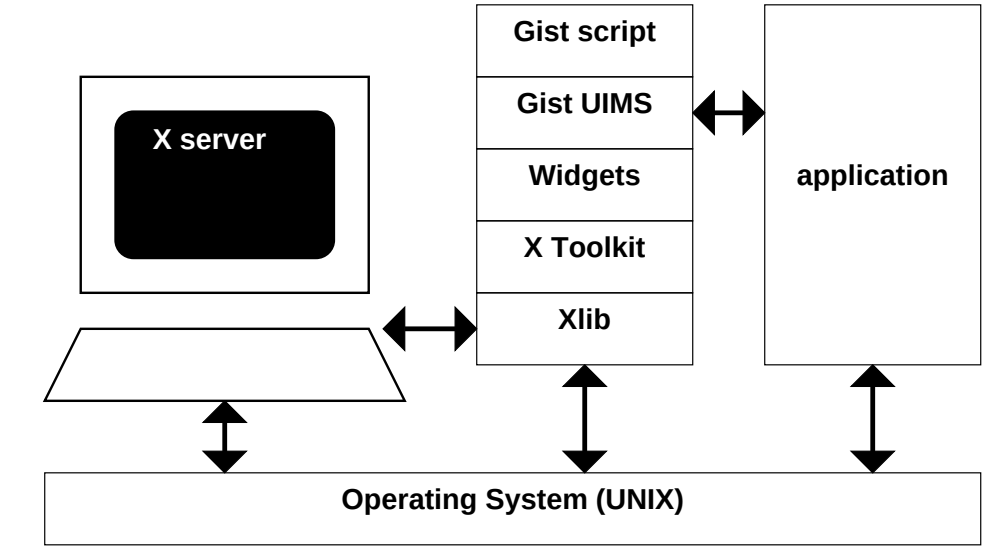


Figure 4.1 The different layers of software that work together in executing an interface that is specified in Gist. Widgets, X Toolkit and Xlib are libraries of the X Window System. When Gist is implemented on other systems, the middle layers in the picture will likely be different.

2. The kernel of the implemented system. This is the system that is described in the next sessions, unless explicitly stated otherwise. The kernel contains the logic to tie interface objects together and to connect an interface to an application, but it does not contain any actual objects. The unresolved issues of (1) are resolved here in an ad-hoc way.
3. A particular installation of Gist on a particular computer. Such a system will consist of the kernel plus many interface elements. It is only with a system like this that you can actually create interfaces.

Gist tries to reach its goals by providing a language for describing interfaces that is based on a mixture of object-oriented principles and Prolog-like recursive procedures. The language is object-oriented in the sense that interface elements are viewed as active objects that do their own processing and that send messages to each other. The way the objects are programmed is reminiscent of Prolog, because pattern matching and recursion take the place of control structures

such as if-then and while-do. A language like this is surveyable and easy to learn, yet concise.

The language is supported by a UIMS that takes care of communicating with the application and that provides debugging, interactive modification, and possibly other aids to the designer. Gist implements a client--server model: the application is the server and is responsible for managing data, performing calculations and the like; Gist is the client, it gives the user a convenient interface to the application.

Some of the unresolved issues have to do with the syntax of Gist scripts: what is the most convenient and least error-sensitive syntax, what is the best notation for the pattern matching expressions. Other issues deal with including or omitting features, such as exception handling, built-in numerical and other operations. Gist also includes an interactive object editor, exactly how that editor should look is also not fixed yet.

4.1 Separation of interface and application

When Gist is used to manage an interface, the interface and the application are two separate programs, run as separate processes. Therefore the implementation language of the application is of no importance, and indeed the application does not even have to be aware of the fact that is connected to a UIMS instead of to a terminal. The approach is similar to that of ToolTool[3]: the standard input, standard output and standard error output of the application are redirected to Gist.

What this means is that the application acts as though it writes text to the terminal and gets input from the keyboard, but unbeknownst to the application, the I/O is intercepted by the UIMS and transformed. It presupposes a multi-tasking operating system, such as UNIX. The phrase «unbeknownst to the application» can be taken literally, but of course it will be much easier to create an interface if the application is designed for the possibility that I/O is to another program. In particular, it helps a lot if the application's output is stateless, in other words: if the meaning of an outputted line of text is not dependent on previous output. The more context information can be extracted from such a line, the less memory the interface itself needs to have.

Even existing applications that are purely text-based and reasonably well-behaved, can thus be fitted with a GUI afterwards. Of course, when the interface and the application are developed together, an efficient protocol can be defined beforehand. All communication between interface and applica-

[3] see Musciano [1988]

tion must be performed by writing strings to the – redirected – standard output and reading strings from the standard input.

Since the protocol is based on text, an interface can also be tested easily, by running it without an application and instead typing the application's text by hand. The interface and the application can each be tested independently of each other.

4.2 Building an interface

Gist does not use direct manipulation. Like FormsVBT, Gist only allows the interface to be changed by editing a script, but unlike FormsVBT the interface can be changed even while it is running. When an interface is displayed on screen, the designer/user can press a special combination of keys – for example, Shift + right mouse button – to pop up an editor window with the script for the object that the mouse points at. The script can be edited. The changes take effect as soon as the editor windows is closed, or even before that, if the «apply» button is pressed. Also, new objects can be created and objects can be removed from the interface. When the session ends, the edited script replaces the original text in the script file.

Interactive editing means that the designer can occasionally paint himself into a corner. More important is the immediate feedback: a modified object changes its behaviour as soon as the «apply» button is pressed. This behaviour is similar to that of HyperCard, when editing HyperTalk scripts that are attached to interface elements. SUIT[4] also exhibits this behaviour.

Currently, that is all that the editor provides, but it should probably be extended with at least on-line help, access to hidden and closed objects and a «what-if» or «undo» capability.

The special key for accessing the editor is governed by a command line switch of the Gist interpreter, because interactive editing need not be available at all times, for example not during normal use of a finished interface.

The script is a normal text file, so it can be edited outside of Gist with any editor. This is probably the preferred way to set up the initial interface and it may also be easier when major changes are to be made to several objects at once, since the script is then loaded as a single text. During a session the pieces of script for each object are edited in separate windows.

[4] see Pausch, Young II and Deline [1991]

4.3 The script language

This section describes the script language of Gist, or rather an approximation of it. The currently implemented language – the one described here – is neither complete nor correct. It is good enough, however, to give an impression of the format and to empirically test the Gist concept in general. The script language should meet the following goals:

- easy to write
- easy to learn
- easy to read
- nice looking
- powerful enough
- easy to document/comment

To meet these goals, the language is designed as follows:

- object-oriented, one object per interface element
- attributes set as keyword-value pairs
- at most one level of indentation
- no control structures, except pattern matching
- objects are programmed with triggers and handlers
- automatic conversion between numbers and strings
- possibility to call external programs

A script is made up of object definitions. Each object has two parts: a set of attributes – such as colour, size, border width – and a set of «handlers». Most of the attributes have a direct influence on what appears on the screen. But Gist does not define attributes or place any limitation on them. Every object defines its own attributes. The precise meaning must therefore be documented in the description of the objects.

A «handler» is a set of actions to be executed when the handler is *triggered*. Handlers are often also called «methods». Handlers can be triggered by user input or by messages that are sent from other objects. Actions can display and hide objects, send messages to objects, create new objects and pass a string to the application. The language has no control structures (functions, conditional statements, repetition statements), at least no explicit ones. It is still possible to program with it, because it supports recursion. People who know Prolog will recognize the style: recursion and matching.

The examples in this chapter are examples that work in a particular installation of Gist. The examples show how a

typical script looks, but they do not imply anything about the availability of particular object types or the meaning of attributes.

4.3.1 User input

Here is an example of a script that creates a window with two buttons,[5] one button is labeled «Cancel», the other is labeled «OK». When activated, the first button sends the string «No» to the application, the other sends the string «Yes».

```
OBJECT window main          # no attribs, use defaults

OBJECT button cancel-button
location "10 10 50 20"      # (x,y,w,h) in pixels
label Cancel
MOUSE-CLICK:  PRINT No.

OBJECT button ok-button
location "70 10 50 20"
label OK
MOUSE-CLICK:  PRINT Yes.
```

The window is named *main*. It has neither attributes nor handlers. The first button is named *cancel-button*. It has an attribute called *location* to set the position and size, a label and a single handler, which is triggered by a mouse click. When it is triggered, it prints «No». Note that there are quotes around the value of the location attribute. They must be used whenever the value includes whitespace, they are optional when the value is a single number or word, like «cancel-button» or «Cancel».

All attributes have default values. The objects in the above example probably have many other attributes such as *color* and *font*.

Usually, the script does not need to define the exact user input, but only the effect it has on other objects or on the application. For example, the button object in the example above also knows how to deal with other inputs such as mouse movement and key presses. These events cause specific forms of highlighting in the button. But only a mouse click has an external effect and is therefore mentioned in the script.

The example also shows that comments can be added to scripts, by prefixing them with `#`. Such comments are ignored by Gist.

[5] We assume that Gist has been installed with objects of type window and button, since, as noted earlier, Gist does not provide any built-in objects. The particular buttons in this example could be implemented with the *XfwfButton* widget from the freely available FWF widget collection, since they it has a *location* resource like the one used here.

4.3.2 Messages among objects

The following example shows how objects can change their own and other objects' attributes. Assume there is no radio-button object available, then one can be simulated with a few buttons and code like the following:

```
OBJECT button b1
location "10 10 50 20"
label "choice 1"
shadow sunken
MOUSE-CLICK:
        self$shadow sunken
        b2$shadow raised
        PRINT 1.

OBJECT button b2
location "10 40 50 20"
label "choice 2"
shadow raised
MOUSE-CLICK:
        self$shadow sunken
        b1$shadow raised
        PRINT 2.
```

This example shows three actions to be executed on a mouse click. The first action changes the value of the object's own *shadow* attribute to *sunken*, the second action changes the *shadow* attribute of the *b2* button. The third acition, PRINT 1, outputs the number 1. The list of actions is terminated with a period.

What happens when these buttons are displayed? They appear in their window at the given location and with the given attributes. Assume they are both active. When the user clicks the mouse on button *b1,* the handler in that button is triggered and the three actions are executed in order. First the button changes its own *shadow* attribute to *sunken*, then it changes the *shadow* attribute of button *b2* to *raised,* and finally it prints the digit 1 (i.e., sends it to the application.)

Actually, this script is an example of bad programming style. The script of one object is allowed to change an attribute of another object. A much cleaner way is to have one object send a message to the other object, which then changes its own attribute:

```
MOUSE-CLICK:
        self$shadow sunken
        b2 raise-yourself   # send message to b2
```

```
        PRINT 1.
...
"raise-yourself":           # handle message
        self$shadow raised.
```

Although I would recommend people to use this cleaner, object-oriented style, Gist does allow the other style, since it is often convenient to be able to make a quick, local, presumably temporary, change.

Messages from one object to another are the main glue that holds the interface together. A handler can be triggered by a literal string or by a pattern, for which handlers provide a regular expression syntax.

Regular expressions have slashes instead of quotes around the trigger. Thus the handler

```
/raise-yourself/: self$shadow raised.
```

would have the same effect as in the previous example, but it would also be triggered by messages that *contain* the text «raise-yourself». The pattern

```
/raise.*yourself/
```

would be triggered by any message that contained «raise» and «yourself» with any number of characters in between.

Currently, Gist uses the POSIX extended syntax for regular expressions. A complete description of that syntax can be found in UNIX manuals under regexp(5). It may be that other notations are easier to use. Compared to some alternatives, such as the syntax used by the Emacs editor or the Lex scanner generator, this seems a good choice, but no attempt has been made sofar to actually devise or test other notations. See section 4.8.2 (page 91) for a discussion of this issue.

The regular expression syntax allows messages with parameters. For example, below is an object that displays a circular progress meter. The *percentage* attribute is a number between 0 and 10000 that determines how much of the circle's area is filled. The handler has a pattern that matches any message that contains the word «set» followed by a space. Everything that comes after the space is saved in a register. Thus, the message «set 5000» triggers the handler and puts «5000» in the first register. The action than sets the *percentage* attribute to the value of that register.

```
OBJECT circperc my-example
/set (.*)/: self$percentage $1.
```

The number of parentheses () determines which register is used: The first pair of parentheses saves to the first register `$1`, the second pair of registers saves to the second register `$2`, etcetera. Every handler has 9 registers, plus the register `$0`, which holds the whole message.

Figure 4.2 The circular progress meter.

4.3.3 Physical and synthetic events

When the user presses a key or moves the mouse, that is a *physical* event. It is received by the interface and dealt with in some way. Most of these events are not handled by Gist, but by the widgets that implement the various objects. The events cause feedback such as highlighting or they change attributes of objects. Only some events cause an effect outside the object. For example, a text field accepts key presses and processes them locally. Only when the Return key is pressed need Gist be invoked. The objects themselves determine when an event has an external effect, thus the events that Gist receives are actually *synthetic* events. Usually this is enough, but Gist provides a type of handler for physical events, that can be used to override an object's own event handling.

There are five synthetic events: *finalize, initialize, mouse-click, mouse-down,* and *mouse-double*. Initialization and finalization actions of objects are attached to handlers for the *finalize* and *initialize* events. They are executed whenever the object is opened or closed, not just when the object is created or destroyed, though for most objects that will amount to the same thing.

The difference between the physical mouse events and the three synthetic events *mouse-down, mouse-click* and *mouse-double*, is that the physical events are delivered directly to the object, and any handler for such an event overrides any built-in reaction to that event. The synthetic events, on the other hand, are generated after a physical event — usually the event after which they are named — is processed by the internal logic of the object. The handler then serves to distribute the results to related objects.

Consider for example an arrow object, such as is found at either side of scrollbars. Such an arrow not only reacts to a mouse button press, but keeps on generating activity as long as the mouse button remains pressed. When the arrow object would be given a handler for the physical event of a button press, that handler would be excecuted only once. But when the handler were instead triggered by the synthetic event *mouse-down*, it would be triggered repeatedly at a certain rate, since this event is generated repeatedly by the object itself.

An additional advantage of the use of synthetic events is that a *mouse-click* event need not be caused by a real mouse click, but can also be the result of an equivalent keyboard event. Simulation of mouse actions with keyboard commands can thus be a part of the objects themselves and need not be dealt with in Gist.

In summary: an object can have handlers for messages, synthetic events and physical events. Messages come from other objects, physical events come directly from the user, and synthetic events are generated by the object itself.

4.3.4 Actions

Three types of actions have already been shown:

- setting attributes,
- sending messages and
- printing. The action *print* sends a text to the application.

Other actions are

- *clone* to dynamically create an object. Such an object is a copy of another object, but with a different name. All the object's children are also cloned, but they keep their original names.
- *open* and *close* to display and hide objects. A closed object is not visible and does not receive events, but otherwise it receives and sends messages just as any other object.
- *activate* and *deactivate* make objects sensitive or insensitive to user events. Usually, objects show themselves grayed when they are insensitive. Inactive objects still receive and handle messages.
- The *halt* action stops the application and the interface. The interface closes down gracefully and saves

any changed scripts, but the application is simply killed. Therefore this action is not often used, except in prototypes.

- *beep* rings the terminal bell.

«Cloning» is similar to «instantiation» as it is used in many object-oriented languages. The difference is that the object that is cloned is a normal object, whereas the class that is instantiated is no more than a template.

An object is cloned together with the objects it contains. These child objects keep their original names, so these names are no longer unique. When an action in a script refers to an object by name, that name is first searched among the children, then among sister objects and finally among the other objects.

Only objects that are defined in the script can be cloned. An object that is itself a clone cannot be cloned further. Also, the clone is a copy of the object as it appears in the script, regardless of how it has changed since the start of the session (static copy).

Clones are useful in situations where it cannot be established beforehand how many objects of a certain type are going to be needed, but the objects are all essentially the same. For example, an editor may offer the user the possibility to open additional windows to view several files at the same time. It would be possible to restrict the number of concurrent windows to some arbitrary maximum, but the cloning facility allows essentially an unlimited number of windows.

Objects are cloned together with their children. This is convenient, because complete windows can be copied in one action, including all the buttons and menus they contain. However, it also introduces a problem, since the names of the children are no longer unique. The solution that is chosen in the current implementation is to search for names among close relatives first: daughters and other descendants, then sisters, nieces, etc. Three other solutions are possible, successively more restrictive than the current one:

1. Allow only references to descendants and ancestors. These objects together form the «scope» of an object. Other objects can only be reached with qualified names, where a qualified name is a name that contains two – or more – parts: the first part is the name of an object that is within the scope, the second part is a name within the scope of the first part.

2. Define the same scope as in (1), but without the possibility of breaking out of the scope with qualified names. To reach an object outside the scope, a message must be sent to an ancestor, asking it to send the message on to the destination object.
3. Define the scope to consist of just the parent and the direct children. Every other object, including grandchildren and ancestors can only be reached by sending messages via a chain of other objects.

Some programmers may like (2) or even (3), but for most people (2) is already too restrictive. The advantage of using one of these three well-defined scopes instead of the fuzzy scope of the current implementation is, that some errors and ambiguous messages can be catched by the system. The disadvantage is that additional concepts and – in the case of (1) – additional syntax have to be learned. The scripts also become larger. It seems to be a question of finding a balance between targeting scripts that are small and easy to write, and scripts that are large, but easier to maintain.

4.3.5 Expressions

Messages need not be constants. There is a simple expression syntax for numerical computations and for combining strings. For example, in the following handler the object decrements its own *value* attribute by one:

```
MOUSE-CLICK: self$value self$value - 1.
```

In this case, the *value* attribute must hold a number, otherwise Gist will complain about an impossible operation. Multiplication (`*`), division (`/`) and addition (`+`) are also possible. Expressions can become quite complex, including parentheses, such as `10 * (self$value/15 + 9)`.

The `+`-operation is special, because it also works for non-numerical values. It tries first to evaluate both sides of the expression as numbers, but if that fails it assumes the user wants to concatenate strings instead. Thus `15 + 7` evaluates to `22`, but `"box" + 7` gives `"box7"`. If you want `15 + 7` to give `157` instead, just start with an empty string: `"" + 15 + 7`, this will force Gist to interpret the expression as a string concatenation.

Gist language summary

Actions

Every interface element (object) can react to user actions and to messages from other objects. In response, it can perform «actions» of the following types:

print ***expression*** The expression is evaluated and the result is given as input to the application. Usually, the expression is a command to the application.

open ***expression*** The expression must evaluate to the name of an existing object. The object thus named is opened (displayed on screen). Nothing happens when the object is already open.

close ***expression*** The opposite of open. The named object is removed from the screen. It is not destroyed, only hidden.

deactivate ***expression*** The expression must give the name of an object. That object will then be made insensitive to user actions. A deactivated object still reacts to messages from other objects, but it ignores all mouse and keyboard events.

activate ***expression*** The opposite. Usually, the change from sensitive (active) to insensitive (inactive) is accompanied by a change in appearrance, but that depends on the type of object.

beep Gives a short beep.

halt Stops the program.

clone ***expr*** **as** ***expr*** Make a new object that is an exact copy of another object. See the discussion in section 4.3.4.

expr**$*****expr*** ***expr*** The first expression must give the name of an object, the second is the name of an attribute of that object. The third expression gives the value that is assigned to the named attribute. E.g., `btn1$label "off"` assigns the text «off» to the label of btn1.

expr expr The first expression must give the name of an object. The value of the second expression is sent as a message to this object. The meaning of a message is determined by the receiving object; there are no predefined messages.

Handlers

Actions are executed in response to a message or an event. «Handlers» determine which actions are executed. A handler consists of a trigger, an optional condition, and the list of actions to execute. Objects can have several handlers, but each must have a different trigger or condition. Example:

```
"pop up" + CLOSED:
    OPEN SELF
    OPEN abc-box.
```

The example shows the **literal text trigger**. It is a text between quotes (") and it says that this handler must be executed when the message «pop up» is received (and the condition +closed is fulfilled, see below).

A trigger can also be a pattern (**regular expression**) instead of a literal, so that it can match with a whole range of messages. Example: `/ab|cd/` is a pattern that is triggered by any message that contains either the letters «ab» or «cd» (or both).

initialize and ***finalize*** are triggered when an object is opened or closed. Example: `INITIALIZE: CLOSE SELF.` This handler closes the object as soon as it is opened.

mouse-down, mouse-click and ***mouse-double*** triggers are normally activated by the corresponding mouse actions. However, the type of object determines when (and if) they are triggered. Some objects, such as static labels, may ignore all user events all-

together. The documentation for each object should describe exactly when these three triggers are activated.

When other events are needed, such as mouse clicks with button 2 or 3, a trigger of the form {***event***} can be used. There is no room here to explain the full syntax of these events, but here are a few examples: {`<Key>Enter`} (triggered by the Enter key), {`Shift<Btn2Down>,<Btn2Up>`} (triggered by a click of mouse button 2 while Shift is down).

By using these events, even objects that normally ignore user events can be made to react to them.

Handlers can be conditional. The example above shows a condition *+closed,* which means that the handler is only executed when the object is closed. Other conditions are *-closed, +active* and *-active*. Triggers of the form {...} cannot have conditions.

The example above also shows the general form of a handler: trigger, optional condition, colon (:), list of actions, terminated with a full stop (.). For clarity, commas may be inserted between actions.

Regular expressions

Gist uses the «extended regular expression syntax» as defined by the POSIX standard. The full definition can be found in most UNIX maunals under regexp(5). Here is a short summary.

Most characters stand for themselves. E.g., `/abc/` matches any messages containing «abc». The characters that have special meaning are:

`.` matches any character, except newline.

`|` separates alternatives, thus `/abc|yz/` matches any message containing either «abc» or «yz» (or both).

`()` are used to group sub-expressions. E.g., `/(ab|yz)23/` matches any message that contains «ab23» or «yz23».

`*` matches zero or more copies of the preceding character or sub-expression. E.g., `/ab*(12)*/` matches messages that contain an a, followed by zero or more b's, followed by zero or more copies of 12, for example «abbb1212».

`+` like `*`, but matches *one* or more times.

`[]` matches one of the characters between the brackets. E.g., `[aeiou]` matches a or e or i or o or u. Ranges can be abbreviated: `[0-9]` matches any digit.

`[^]` matches any character *except* those between the brackets. E.g., `[^A-Za-z]` matches any character that is not a letter.

`^` if placed at the beginning of a pattern, anchors the pattern to the start of the message. `/^do/` matches any message that starts with «do».

`$` if placed at the end of a pattern, anchors the pattern to the end of the message. `/^ab.*yz$/` matches any message that starts with «ab» and ends with «yz».

The special characters loose their special meaning inside []. The combination `\t` can be used instead of the tab character, which would otherwise be invisible.

Attributes

All objects have a set of attributes that determines how the object looks and sometimes also how it behaves. Which attributes are present depends on the type of object, but typical attributes are *background* (for the colour of the object's background), *font*

(for the name of a font), *x* and *y* (for the position relative to the object's parent). Here is an example:

```
OBJECT button ax-27 # define "ax-27"
  x 10              # x position 10
  y 100             # y position 100
  label "stop now"
  background blue
```

The quotes around the value are needed when the value includes whitespace, as in `"stop now"`. They are optional around numbers and single words, as in `"blue"`.

Objects

Every object starts with the word *object,* upper or lower case doesn't matter. Next comes the type, the name, an optional parent, optionally the word *closed*, then zero or more attributes and finally zero or more handlers. Example:

```
OBJECT icon bell (window1) CLOSED
x     25
y     5
icon  "bell.xpm"
MOUSE-CLICK:
    BEEP.       # Beep on mouse click
```

The parent can be omitted if the object is to be a child of the window or box that precedes it in the script file. The addition of *closed* says that the object is initially closed (hidden from view). Objects without this flag are opened at the start of a session.

The type must be one of the types that was configured when Gist was created. Usually there will be types such as window, box, button, icon, label, etc.

The name must be unique within the script.

Global defaults

Global defaults look just like attributes, but they are not attached to any object. Instead they are remembered and attached to all objects that do not explicitly state a different value. Global defaults must be written at the start of the script file, before the input model (see below) and before any objects are defined.

Note that the standard resource mechanism of the X Window System may interfere with the global defaults. See the explanation in 4.5.

Miscellaneous

The input model is a set of handlers that is inserted after the global defaults, but before the first object. There must be a handler for every (meaningful) line of output that the application prints.

The syntax of expressions is explained in 4.3.5.

4.3.6 Calling external programs

Gist provides a way to start external programs as a side effect of actions, but with some limitations. When the program is started, Gist waits for it to finish before continuing. The program's output is collected into a single message and it can be sent to any object or to the application. Since the system is effectively halted while the external program executes, this facility should only be used for programs that perform some computation and then exit again quickly.

Experience has shown that this capability is sometimes needed for computations that Gist itself cannot perform. Whether this limited support is enough is still a question for future research. Letting an external program run asynchronously poses some problems, but there are ways in which parallel programs can be synchronized in better ways than is currently the case (like in Awk, for example.)

4.3.7 An example that uses cloning

The following example shows a first approximation to an E-mail interface. It has a main window with icons for all the mail folders, including two pseudo-folders (waste basket, and letter-box). There is a menubar at the top of the window. Double clicking on a folder opens a new window, containing icons for all the messages in that folder. Double clicking on a message icon opens yet a third kind of window, that shows the contents of the message. There are some buttons along the bottom of this window.

Since the user may open as many folders and messages as he likes, the corresponding windows must be created dynamically. The script therefore defines three windows: the main window, of which only a single copy exists, a folder window that will be cloned as often as the user double clicks on a folder icon, and a message window, that will be cloned everytime the user double clicks on a message icon.

We assume that the E-mail application has been specially written to accept the commands that the interface defines and to print output in a format that is easy to parse. Since this is just an example, many parts of the interface and of the protocol between interface and application will be left unspecified. The input translation for messages from the application is not shown, but it should be able to do things such as: translate a list of folders to the format expected by the iconbox, and translate a list of messages within a folder to a similar list.

As in the previous examples, we assume the existence of suitable objects, in this case a menu bar, a pulldown menu, an icon box, a normal button, a row/column object and a message viewer. The iconbox object has an attribute *list* that contains a list of icons and labels. When this attribute is set, the icon box creates labeled icons and manages them. A double click on one of them is translated to a message of the form «open...», drag and drop operations are translated to messages of the form «on *n* from *icon-box* drop...». Icons can be dragged from one iconbox to another. The «from» part gives the name of the object from which icons are dragged.

The main window has two parts: a menu bar and an icon box with icons for all mail folders. The menu bar contains drop-down menus, of which only a few are shown.[6]

```
OBJECT window "Example E-mail interface"

OBJECT menubar main-menubar
location "0 0 1.0 30"

OBJECT menubutton folder-menu (main-menubar)  # Drop-down menu
label "Folder"
menu "New,new;\
   Open,open;\
   Delete,delete;\
   Exit,exit"
# ... plus some handlers

OBJECT menubutton help-menu (main-menubar)  # Drop-down menu
label "Help"
menu "Help,help"
# ... plus handlers

OBJECT iconbox folders
location "0 30 1.0 1.0-30"
initialize: PRINT "list\n".      # Ask appl. for list of folders
/list (.*)/: self$list $1.       # Msg. from input: create icons
/open 1:.*/: .                   # Dbl. click icon 1 (=letter box)
/open (.*)/:                     # Dbl. click any other folder
   DELETE $1                     #  Delete if it existed already
   CLONE aFolder AS $1.          #  Create window for folder
/on - from (.*) drop (.*)/: .    # Drop on background, ignore
/on 0 from folders drop (.*)/:   # Folders on 0 (=waste basket)
   PRINT "delfolder "+$1+"\n"    #  Delete a folder
   PRINT "list\n".               #  Update list of folders
/on 0 from (.*) drop (.*):       # Drop msgs on 0 (=waste basket)
   PRINT "folder "+$1+"\n"       #  Go to folder
   PRINT "del "+$2+"\n"          #  Delete messages
   PRINT "list "+$1+"\n".        #  Update list of messages
/on .* from folders drop .*/: .  # Drop folders, ignore
/on 1 from (.*) drop (.*)/:      # Drop msgs on 1 (=letter box)
   PRINT "folder"+$1+"\n"        #  Go to folder
```

[6] The *location* attribute that is used in many examples in this chapter is a concise way of specifying position and size in a single attribute. It has four parts: x coordinate, y coordinate, width, and height. Each of these may have an absolute and a relative value. E.g., `"0.25 10 0.5 1.0-20"` says that the object is located at 0.25 of the width of its parent, 10 pixels from the top, it is half as width (0.5) and 20 pixels less tall than its parent (1.0-20).

```
   PRINT "send "+$2+"\n".          #  Send selected messages
/on (.*) from (.*) drop (.*)/:     # Drop messages on a folder
   PRINT "folder "+$2+"\n"         #  Go to folder
   PRINT "move "+$3+" "+$1+"\n"    #  Move msgs to other folder
   PRINT "list "+$1+"\n"           #  Update list of folders
   PRINT "list "+$2+"\n".          #  Update list of folders
```

Note that the order of handlers is important. Messages involving the letter box (icon 0) and the waste basket (icon 1) are defined before the general case of the normal folders.

The window that is cloned when the user double clicks on a folder icon is defined next. It has an icon box similar to the one above, but this time it contains icons for messages. There is a row of buttons along the bottom of the window.

```
OBJECT window aFolder

OBJECT iconbox messages
location "0 0 1.0 1.0-40"
initialize:                        # Executed when opened
   PRINT "list "+self$name+"\n". #  Ask appl. for list of msgs
/list (.*)/: self$list $1.         # Msg. from input, create icons
/open (.*)/:                       # Dbl. click on an icon
   DELETE "["+parent$name+"]"+$1 #  Delete if it exists already
   CLONE aViewer AS "["+parent$name+"]"+$1. # Create new viewer
/on .* from folders drop .*/:  . # Drop folders, ignore
/on - from (.*) drop (.*)/:        # Drop on background
   PRINT "folder "+$1+"\n"         #  Go to folder
   PRINT "move "+$2+" "+parent$name+"\n" # Move msgs to here
   PRINT "list "+parent$name+"\n" #  Update list of messages
   PRINT "list "+$2+"\n".          #  Update list of messages

OBJECT row-column button-bar
location "0 1.0-40 1.0 40"
"close-window":                    # Message from close button
   DELETE parent.
"view-msg":                        # Message from view button
   parent "open "+parent$selected.
"delete-msg":                      # Message from delete button
   folders "on 0 from "+parent$name+" drop "+parent$selected.

OBJECT button view-btn (button-bar)
label "View"
MOUSE-CLICK: parent view-msg.

OBJECT button delete-btn (button-bar)
label "Delete"
MOUSE-CLICK: parent delete-msg.

OBJECT button close-btn (button-bar)
label "Close folder"
MOUSE-CLICK: parent close-window.
```

The next defined object should be the window *aViewer,* that is cloned in the actions above. It is omitted here, since it

introduces nothing new. There should also be more buttons and more menu entries.

The example has a few handlers that do nothing at all. For example, when the user tries to open the letter box – icon 1 in the *folders* icon box – nothing happens. These handlers are, in fact, error handlers, since they catch incorrect user actions. Instead of doing nothing, they could also sound a beep or pop up a dialog box.

When the protocol between application and interface is fully defined, it will probably also include error messages. The interface should then define ways of dealing with these application errors as well.

4.4 Modelling the application

When the interface is constructed outside the application program, there needs to be a way of connecting the two. Applications that use a toolkit do not have this problem, since the interface is part of the program; neither do systems like HyperCard and Matrix Layout that build the whole application themselves. In general there are three possibilities:

- Let the application do the translation. This means that the interface is – perhaps implicitly – described inside the application either as a tree of objects (e.g., ResEdit under GEM) or as a set of procedures (e.g., Dirt, Wcl, FormsVBT).
- Create an intermediary. A separate program could translate between the I/O of the application and the actions in the interface. This approach is taken by none of the systems.[7]
- Integrate the translator into the UIMS. This is done in ToolTool and DIGIS and it is also the approach taken by Gist. It allows for a more flexible interface, which is, after all, the part that is more likely to require structural changes than the application, both during development and afterwards.

The description of the application can take various forms. In DIGIS it is a complete formal definition of the semantics of the application in a special, object-oriented language. Not only the output of the application is described, but a model is defined that describes the various states the application can be in. Of course, only the states that are relevant to the GUI need to be described.

[7] But compare Smalltalk's Model-View-Controller paradigm (MVC) and the Semantics-Appearance-Interaction triplet of MoDE (Shan [1991]) that is derived from MVC. The Controller, eq. the Interaction part, acts as the intermediary. The three parts are not separate programs in the traditional sense, but objects in an object-oriented operating system, viz. Smalltalk. There is also not a single controller, but as many controllers as there are interface elements (views).

In ToolTool and in Gist only the application output is described explicitly. Application input and application states — insofar as needed — are implicit in the general working of the interface.

Gist uses the same handlers as are used to determine the behaviour of objects to define a virtual, unnamed input object. The handlers of this object define the semantics of the application's output by the list of actions they execute when they are triggered. In this way, any text that is output by the application results in attributes being set and messages being sent to objects. The list of handlers must be inserted into the script before the first real object.

Since the input object is not a real object, it has no real attributes. Instead, anything that looks like an attribute will dynamically be created as such. In other words, the input object provides variables that do not need to be declared. Like real attributes, these variables only contain strings, but they will be converted back and forth to numbers when they appear in arithmetic expressions.

The variables can be used to remember the state of the application between outputs.

4.5 Setting global defaults

It is often useful to set global defaults for some attributes. Attributes such as typeface and colour are often better expressed with defaults and local exceptions. Default values for attributes are entered in the script even before the input handlers.

Attributes can thus be set globally or for individual objects, but there is no way to specify default values for attributes of a particular type of object (all buttons, say) or for a limited set of objects (e.g., all objects in window A). This is an aspect of Gist that yet has to be solved.

There are two reasons why one would want to set attributes for a group of objects as a whole:

- for convenience: for example, all labels in one window should be blue, while all labels in another one should be green. If there are many labels in each window, it is convenient — and easier to change — if the colours can be specified in one place only. Note that there are other ways to do this, such as symbolic constants or macros. Neither of these is provided by Gist, but they should be considered if Gist is extended.

- for consistency across applications: a user may want to set fonts, colours, etc. for all similar elements in all applications on his system. An interface created with Gist should then conform to these system-wide attributes as much as possible.

Gist tries to be a well-behaved client of the X Window System, so it leaves the latter to the resource manager that is part of X. Every implementation of X provides ways to set resources globally for all programs that use the same display (although not always in a convenient way).[7]

When Gist is implemented on platforms other than X, a different solution will have to be found, preferrably one that interacts well with the environment. If the platform provides a standard tool for setting attributes across applications, Gist should of course not interfere. But if there is no such tool or if the tool is not powerful enough, the Gist syntax may have to be extended.

In any case, this means that the interface may look (or even behave) differently depending on resource settings over which Gist has no control. The only way to have absolute control would be to specify *all* attributes of all objects and not to rely on defaults at all. But would you want to do that? When a user has configured his workstation in a certain way, should any interface try to do things differently?

4.6 Configuring & extending Gist

Since Gist provides just the glue that holds an interface together and not the actual interface elements, any installation of Gist has to be configured for some external set of objects. Likewise, when a new object is to be added, additional configuration has to be done. This configuration can only be done by a programmer.

The configuration is done with a special language (see section 5.10). The configuration file fully describes each type of object. It lists all attributes and defines when the synthetic mouse events are generated. It also defines how the object maps to the widget that is used to implement it. After the configuration is changed, Gist must be recompiled.

4.7 Portability

The Gist project started in 1988 with a prototype on a PC under MS-DOS. Since MS-DOS has no multitasking, the system was set up as a library to be linked to an application. When it moved to UNIX and X, the application and

[7] Incidentally, the resource manager of X can also 'solve' the first problem. With revision X11R5 there is even an utility called «Editres» for setting attributes interactively. It works under revision X11R4 as well, but only on conforming applications.

the interface could be decoupled as they are now, relying on interprocess communication instead of on function calls. The syntax of the script remained essentially the same.

The current implementation relies heavily on the facilities provided by X and the X Toolkit. It should be possible to port Gist to any system that runs UNIX and X. It would be a little harder to port it to a system that runs X on anything else than UNIX, since a new solution has to be found for the interprocess communication. Porting to a system without X will be virtually impossible. Except for the script syntax everything will have to be rewritten.

Does this mean that Gist is not portable to anything else than UNIX workstations running X? Yes and no. The implementation is not portable, but the user interface (the script) is. The situation is no different from that of programming languages: the compiler or interpreter is not portable, but the language itself is.

For more – and more technical – information, I refer the reader to next chapter and to the syntax and the example interface in the appendices.

4.8 Advantages, disadvantages, possible enhancements

Clearly, Gist is not a complete solution to the problems of interface designers. It supports prototyping and building of the final interface, but it can play no role in the earlier stages when information is gathered and ideas are discussed. Gist also depends on the availability of a suitable collection of interface elements. New elements can only be created by programmers. On the other hand, when new objects become available, they can be added to the system permanently and will be indistinguishable from older elements.

The reliance on external objects also has an effect on the amount of control that can be excerted over an interface with the facilities of Gist. Earlier it has already been noted that Gist cannot control the keyboard focus. Other aspects that it delegates to the objects themselves or to the environment – i.e., the X Window System and the window manager – are for example: mouse acceleration and double click delay, drag and drop capabilities, resizing and moving of windows or other objects.

4.8.1 Coupling of application and interface

The way the interface and the application are coupled has the advantage that the application can be written in any language

and can be debugged independent of the interface. Well-behaved existing applications can also be accomodated. The interface can likewise be tested independently. Since only text is exchanged between application and interface the protocol can be checked and even simulated by hand.

On the other hand, there is a penalty in speed. When a lot of very detailed information has to be passed from the application to the interface, the overhead of conversions to and from text and the management of the channel between the two programs may be a problem.

A solution may be to give the application the possibility to draw directly to the screen, bypassing the interface, but only in a reserved area or «canvas» provided for this purpose by Gist.[8]

[8] This approach is used, for example, in the Ghostview interface to Ghostscript, a PostScript interpreter. Ghostview builds an interface which includes an empty canvas. It then starts Ghostscript and passes it a handler (identification number) to the canvas. Ghostscript can assume the canvas has already been created and starts drawing right away.

4.8.2 Script language

To a certain extent, programming languages are a matter of taste. Of course they should provide concepts at the right level of abstraction and have a syntax that is easily learned and not very susceptible to mistakes, but aspects such as the opportunities for an esthetical lay-out or similarities to familiar languages[9] are also important.

To non-programmers ease of learning is important, but also the underlying programming paradigm. Programmers are familiar with algorithmic solutions, but non-programmers often prefer declarative techniques.

The scipt langauge of Gist tries to be as static as possible, with as little syntactic overhead as possible. The attributes are simple keyword value pairs. The dynamics are specified as handlers to be triggered by certain events. They do not execute a program, but a simple linear list of actions and they do not store data in characters, integers, arrays or pointers, but only in strings.

Users will most likely develop their own set of «tricks» to perform what programmers would recognize as conditional execution, repetition or recursion. These tricks will depend on manipulation of strings and regular expressions and will therefore not be as efficient as the same actions done with a more structured programming language. This is not a real problem, since the scripts for each object will be small.

The expression syntax of Gist is rather limited. Although an interface typically does not do much computation, it is probably a good idea to extend the syntax with more operations, including some built-in functions such as *trunc* or *round*.

[9] The influence of the C language is apparent in this matter. Motif UIL is very similar to C in appearance, even though it provides only a purely static description of an interface. In other areas C also demands its toll, an example is the C Shell, which is clearly inferior to most other command interpreters for UNIX, but often preferred by programmers because of its superficial similarity to C.

The syntax of the regular expressions is perhaps not the most user-friendly, but at least it is the same syntax as used by other UNIX utilities. However, that may not be an argument for many users, who are not familiar with the programs that use this syntax. One of the things that should therefore be investigated is whether a restricted syntax would be easier, without losing expressive power. A syntax based on wildcards might be good compromise. A wildcard syntax like that used by many command interpreters for matching filenames is not as powerful as a full regular language, but it may well be powerful enough.

Also, currently patterns are not anchored. That means the pattern can match a part of a message, it does not have to match all of it. For example

```
/[0-9]*/
```

would match *every* message, since every message contains zero or more digits. To anchor the pattern, it would have to be written as

```
/^[0-9]*$/
```

The `^` matches the start of the message, the `$` anchors the pattern to the end of the message. Now only empty messages or messages that consist entirely of digits will match.

Experience so far seems to suggest that anchored patterns are more convenient, so it is likely that in the future anchored patterns will become the default, obviating the need for `^` and `$`.

4.8.3 On-line help & error handling

Gist has no facilities for on-line help. Since on-line help is very important, this is a major omission. Some objects may have their own methods for providing help (for example, Motif widgets have some support for this). The main reason that there is no mechanism for on-line help in Gist is, that it is not clear what it should be like: should it be built-in to the widgets, as in Motif, or should it be a separate mechanism; should the help text – which need not be just text – be in the script file or in a separate file.

Error handling also is not handled specially by Gist. Errors caused by incorrect user input can be recognized at different levels in the processing. An object can have its own internal rules, such as that a postal code (zip code) has exactly 6 letters/digits. Incorrect input of this type is handled at the

object level. Semantic errors are handled by the application, for example, typing an incorrect password is detected by the application, not the interface. Between the two there may be a third type of errors, presumably errors that have to do with the relations among interface elements, that cannot be dealt with by individual objects and yet are not of a semantic nature.

In a well designed interface such errors are impossible or at least rare. The script language is powerful enough to detect such errors and therefore there is no need for a separate exception handling mechanism. To be more precise, I have not been able to think of sufficient examples of such situations to warrant a special treatment.

4.8.4 Interactive design

Gist doesn't use direct manipulation to set up an interface, although that could be a more convenient method, especially in the early stages of the design, when objects are initially placed on the screen. When positions, sizes and other attributes must be specified more precisely – including size constraints between objects – direct manipulation soon becomes cumbersome.

An advantage of interactive design, whether by direct manipulation or not, is that is becomes possible to do easy «what-if» experiments. Changes can be made temporarily and only committed when they live up to expectations.

On the other hand, pure interactive design makes it hard to document and store intermediate versions, since they are usually not kept in an accessible format. Script-based systems and mixed systems like Gist have an advantage here. The scripts can be printed, annotated and stored as documents.

4.8.5 Possible enhancements

There are many things that could be improved in Gist, yet do not change the overall structure. A few are listed here:

Property sheets

When interactively editing an object, it might be convenient to have a list of the available attributes on-screen, instead of in a paper manual. The system already has access to this information and it would be an easy matter of making it available to the user.

Tracing, timing, logging

During the devolpment of an interface it is often useful to have records of how users interact with it. So-called «protocols»

are exact logs of what happened between the user and the interface at what moment in time. Gist could be extended to – optionally – provide a typescript of a session. Post-processing tools would be needed to massage the raw data into something more managable.

Demo mode

Systems that employ a lot of graphics are often hard to describe. To teach somebody to use it, a demonstration is often a better method. A demonstration session can be recorded on video, but it would be much simpler if the interface could demonstrate itself. A facility for playing back session could be added, maybe by feeding back the typescript mentioned above.

Chapter 5

The implementation of Gist

Gist is not very large. It contains a parser for scripts, an editor for interactively changing the interface, an interpreter that can execute the actions in the script and a handful of routines for enhancing widgets to work with Gist. The X Toolkit, with which it is linked, provides the routines to deal with low-level events.

Gist is independent of the actual interface elements (the widgets) used, although any concrete implementation must of course be compiled with a particular set of widgets. To ease the incorporation of widgets as interface elements under the control of Gist, a utility is provided that converts an editable configuration file into a module that can be linked with the rest of the program.

Figure 5.1 shows the different modules from which the program is built. The modules are explained in some detail in the sections below.

5.1 Flow of control

The program works under the X Window System and is therefore a client of the X server. Any interaction with the screen, the mouse and the keyboard is done by the X server, in reaction to requests from Gist.

If the command line options have been removed, the first argument on the command line is the the name of the file that contains the interface description. That file is then parsed and widget instances are created by the parser. If there were any errors, the program stops at the end of the parse. If not, and there was a second command line argument, that second argument is interpreted as the application to execute. The application is forked, with its standard input and output redirected to a pty (pseudo-tty). The pty is placed under the

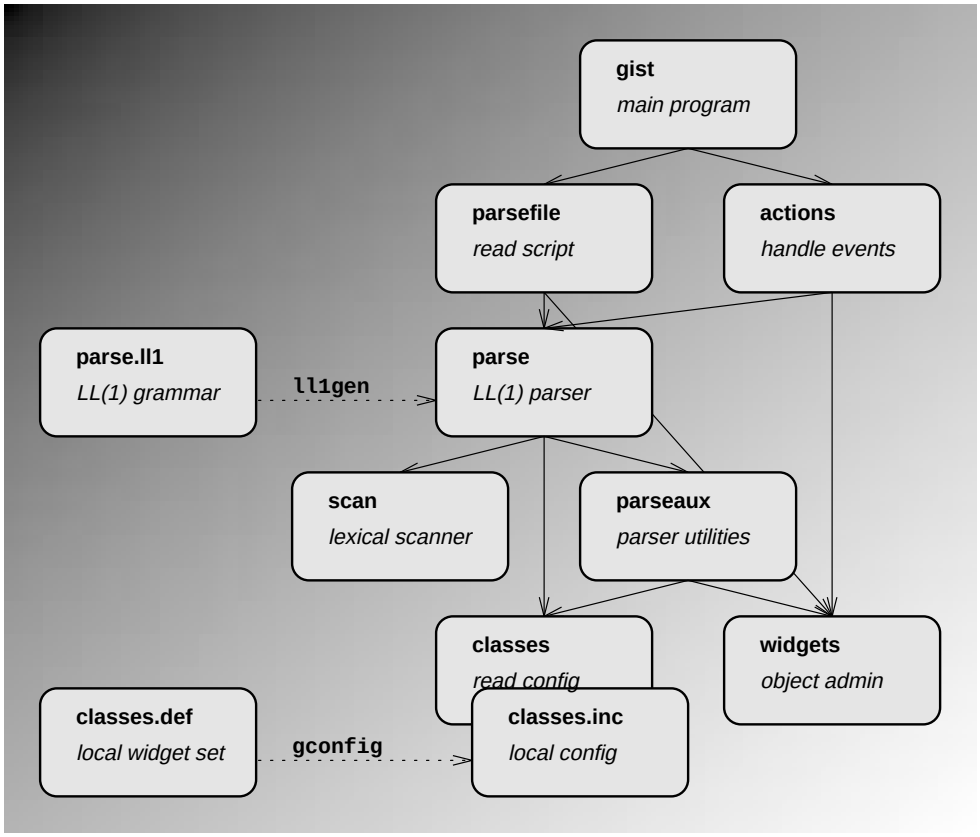


Figure 5.1 Gist is built up from a number of modules. The «classes.def» module is installation dependent, since it contains the code to connect specific widgets to the system. It is automatically generated from a local configuration file (not shown).

control of the X Toolkit, so that Gist is informed whenever the application writes something.

The program then initializes the widgets, starts the main event loop and waits for things to happen.

The program relies heavily on the functions provided by the X Toolkit. In some places this has necessitated conversions of data to and from the format expected by the X Toolkit, in some cases it has even been necessary to use strings as the lowest common denominator. Such conversions are performed only once, however. On the other hand the use of the toolkit helps much in making the program more reliable and easier to maintain. It also makes the use of widgets possible.

Overall the use of the toolkit seems to have more advantages than disadvantages, as compared to working without it.

When the user causes an event, or when the application produces output, Gist will become active, because the X Toolkit calls a Gist function. Gist looks up the appropriate handler for the event and executes it, maybe causing a whole string of actions and other handlers to be executed as well. But eventually the actions will stop and Gist will become dormant again, until it is activated by the X Toolkit to handle the next event. The routines that are involved in this are contained in the «actions» module, see below.

The program stops either when it encounters the «halt» action in one of the handlers or when the application dies.[1] The interface objects are destroyed and if the interface has been changed, a new script is written to make the changes permanent.

5.2 Datastructures

The Gist program has to keep lists of classes and objects and for each object also a list of handlers. Handlers in turn contain a list of actions and actions contain expressions. The data structures that hold this information are used throughout the program and they are defined in a header file «types.h», which is included by every module.

All string constants are hashed, with the help of some routines provided by Xlib. The hashed string is of type *XrmQuark,* which is a type defined by the X libraries.

Expressions are either constant strings or binary operations, which may be nested. If the expressions are binary operations, the operands are interpreted as numbers, i.e., the strings at each side may only contain digits and whitespace. Gist will display an error message otherwise. An exception is the *AddOp*, which will fall back to string concatenation and not give an error message.

```
typedef enum {AddOp, SubOp, MulOp, DivOp} Operation;
```

The *Expression* type is the union of a *LeafNode* and a binary expression (*Node*). The boolean field *leaf* indicates which is which. The *LeafNode* can be either a literal string, a shell command, a reference to an attribute, or a reference to part of the message that triggered the action. The combination of fields determines which.

```
typedef struct {
    Boolean leaf; /* = always True */
```

[1] Gist doesn't check whether the application died with a non-zero return code. This could be added as an option, but currently Gist relies on the application printing an error message just before it exits.

```
    Boolean shell_cmd;
    XrmQuark v, sub;
} LeafNode;

typedef struct {
    Boolean leaf; /* = always False */
    Operation op;
    union _Expression *left, *right;
} Node;

typedef union _Expression {
    Boolean leaf;
    LeafNode leafnode;
    Node node;
} *Expression;
```

There are nine types of actions (an action that destroys a dynamically created object has not yet been implemented) and there is a struct *Action* with enough fields to hold the information for each of them; depending on the type of action, some fields remain unused or have different functions.

```
typedef enum {AClone, AOpen, AClose, APrint, ASend,
    AHalt, ABeep, AActivate, ADeactivate} ActionType;

typedef struct _Action {
    ActionType tp;
    Expression receiver;  /* receiver or name */
    Expression sub;       /* attribute or NULL */
    Expression value;     /* class, value or NULL */
    Boolean closed;       /* only if ANew */
    struct _Action *next;
} *Action;
```

The field *receiver* is used when the action type is *AClone, AOpen, AClose, AActivate, ADeactivate* or *ASend*. It holds the name of the object to clone, open, close, activate or send a message to. The field *sub* is only used with the *ASend* action, if the action is not to send a message, but to set the value of an attribute. It holds the name of the attribute The field *value* is used with the *ASend* action to hold the message to send or the value to set in an attribute and it is used with the *AClone* action to hold the name of the object to create.

Handlers are differentiated by the type of their triggers:

```
typedef enum {HString, HRegexp, HEvent, HFinal,
    HInit, HMouseDown, HMouseClick,
    HMouseDouble} HandlerType;
```

A handler can be made conditional, which means that it is only executed when the object is open, closed, sensitive or insensitive. The fields *closed* and *sensitive* record the conditions.

```
typedef enum {IfTrue, IfFalse, IfAny} ThreeValue;

typedef struct {
    HandlerType tp;
    union {
        XrmQuark string;
        regex_t re;
        XrmQuark event;
    } trigger;
    ThreeValue closed;
    ThreeValue sensitive;
    Action actions;
} Handler;
```

Handlers of type *HString* are excuted if the object receives a message that is equal to *trigger.string; HRegexp* handlers messages that match *trigger.re; HEvent* handlers execute if the event *trigger.event* occurs (the event is coded in the X Toolkit syntax). The other types of handlers react to events that are implicit in their type: *HFinal* is executed on every «unmap» event, *HInit* with every «map» event, *HMouseDown, HMouseClick* and *HMouseDouble* are synthetic events, generated by the objects themselves, but usually in response to a press, a click or a double click of mouse button 1.

The configuration file that lists the available classes of objects also indicates whether the object is to be used as a window or dialog box instead of a normal element. This is important, since windows and boxes do not have parents within the interface, but instead have a shell widget created for them. (See also «classes» below.)

```
typedef enum {MainWindow, DialogBox,
    NoShell} ShellType;
```

The parser builds a list of widgets. The top level widgets (windows and boxes) are not opened by the parser, instead a flag is set in the list. At the end of the initialization the main program opens all widgets that have this flag set. The *script* field is a copy of the part of the script file that defines this widget.

```
typedef struct {
    Widget widget;
```

```
    XrmQuark class;
    Boolean open;
    String script;
} WidgetDesc;
```

5.3 The «main» module

The module «main» contains the start-up code and it also contains the routine that receives the output from the application. The latter routine could have been put in a separate module, but since it is quite short and self-contained it has been left in the main module.

The main routine performs a number of initializations. If one of them returns an error, the program will be aborted:

- Open a connection to the X display server. The X Toolkit provides the routine *XtVaAppInitialize* for this task.
- Initialize the set of widget classes. This is done by a routine in the «classes» module. The effect is that all class and attribute names are hashed, for faster access later.
- Create a pseudo-object *inputwidget* that will be used to represent the application in communications among objects.
- Check that there is a command line argument. This argument is the name of the script file. The file will be parsed, resulting in a collection of widgets. (See «parsefile» below.)
- If there is a second command line argument, this is the command to start the application. A «pty» (see below) is created and the application is forked, with its input and output redirected to the pty.
- The final initialization step consists of the «mapping» of all top level windows, i.e., opening all windows that should be open at the start of the program.

The main routine ends with a call to *XtMainLoop*. This X Toolkit routine contains an infinite loop that waits for events and calls the appropriate routines in Gist.

When the interface is used with an application, the application is started as a separate process (forked) and its input and output are redirected to a pty. A pty (or *pseudo-terminal*) is similar to a two-way pipe or a socket, but the advantage of a pty is that it looks like a normal terminal to the application. Many programs will detect whether they are outputting to a terminal or not and use buffered I/O if it is not a terminal.

Buffered I/O disables interactive work, for to work interactively with an application, Gist must see every line of output as soon as it is generated and the application must also react to every line of input immediately. Pty's seem to be the only way to force this behaviour between two processes.[2]

Pty's are opened like any other device or file, but there is only a limited number of them available (often 26). The standard way of obtaining a pty from the system is simply to open them all in turn, until one is successfully opened or until the list is exhausted. In theory it is therefore possible that Gist will fail because all pty's are in use by other processes. In practice this is seldom a problem.

5.4 The «parsefile» module

The module «parsefile» is only used during initialization. Its task is to read the script file, pass it to the parser and store the objects (widgets) that the parser creates. The objects are parsed one at a time, since this allows the same parser to be used both for reading the script and for changing an object after it has been edited interactively.

A utility routine *read_up_to_object* is used to read the complete description of an object into a string, which is then handed to the lexical scanner. When the lexical scanner has thus been initialized, the parser is started. This process is repeated until all objects have been read. If any of the calls to the parser caused an error message, the function returns *False,* causing the main module to abort the program.

Here are the relevant lines:

```
while ((s = read_up_to_object(f, last))) {
    init_scan(s, 0);
    ll1parse(&widget, &classname, &closed, toplevel,
             &default_parent);
    store_widget(widget, classname, !closed, s);
    end_scan(&dummy);
}
```

To the user this piecemeal parsing is hardly noticable. The only effect is that the script is not completely free-form as it would have been if it had been parsed in whole. The exception is the keyword OBJECT, which must be the first word on its line for the parsefile module to find it.

The parser returns the widget that it has created. The widget is stored by the module «widgets» (see below). The information that is stored consists of the object's name, class, handlers and whether it is open or closed. All this information is returned in the parser's parameters.

[2] Not all versions of UNIX provide pty's, but there exists a fairly portable freeware implementation.

The lines before the first object in the script contain default attributes and describe a model of the application, in the form of handlers for each of the possible (and meaningful) outputs of the application. These first lines are parsed as if they were the declaration of the pseudo-object *inputwidget* (see the main module above).

5.5 The «scan» module

The lexical scanner is a simple, hand-made module, with as most important function *nextsym*. The scanner is initialized by calling *init_scan* with a string as argument. The next token is stored in a global variable *sym* and every call to *nextsym* changes it to the next token. If the token is an identifier, string or something similar, the actual string is stored in another global variable, called *curtoken*.

The module exports two other functions: *end_scan* and *scan_status*. Both can be used to retrieve the position in the string where the last recognized token started. *scan_status* also returns the string that was passed to *init_scan*. The error routines in «parseaux» (see below) use this to show the position of the error to the user.

The algorithm to recognize keywords and identifiers relies on the Xlib routines that hash strings into «quarks». The first time *init_scan* is called, it creates the quarks for all keywords. All identifiers that are encountered in the input are then hashed and their quarks are compared to the quarks for the keywords. Attributes, messages, names of objects, etc. are stored as quarks, so it is convenient to use them for keywords also, even if it provides no gain in speed.

The lexical scanner is also responsible for skipping comments. Comments in Gist scripts start with a '#' and go on for the rest of the line. This is the same convention that is used in UNIX shell scripts, and therefore it allows the scripts to be made executable. If the script starts with the comment

```
#!/usr/local/bin/gist
```

(or whatever the full path name is) the script can be executed, just like a shell script.

5.6 The «parse» module

The actual parser is generated from a grammar with the help of a parser generator, called «ll1gen». As the name implies, ll1gen is a parser generator for LL(1) grammars, the generated parser is therefore a recursive descent parser.

The obvious choice would have been to use «yacc», the standard UNIX parser generator, but tests and experience with other yacc-generated parsers have led to a decision to generate a top-down parser instead. The error messages that a bottom-up parser is able to produce for syntax errors are usually not very informative and it was felt that the quality of error messages was particularly important. With a top-down parser good error messages are possible, provided the parser has a reasonably sophisticated error-recovery strategy.

The complete syntax can be found in an appendix.

5.7 The «parseaux» module

The routines in the module «parseaux» perform miscellaneous functions for the parser. They are:

error	*check_or_create_parent*
manage_maybe	*eval_const*
new_node	*new_leaf*
process_handlers	*set_default_attribute*
set_attribute	

The *error* function is used to display errors. It pops up a dialog box to show both the context of the error and a description.

The routine *check_or_create_parent* has three functions, depending on the type of object it is called with. If the object is classified as a *MainWindow* or *DialogBox,* it should not have a parent object. Instead a shell widget is created for it. The object is also marked as the default parent for subsequent objects.

If the object is not meant to be used as a window or box, it must either have an explicit parent, or it will be assigned the current default. It can happen that there is not yet a default parent, which means that there is an error in the script. In this case the error function will be called.

The function *manage_maybe* «manages» (opens) an object, unless its default state is closed or it is a window or box. In the case of windows and boxes a flag is passed back to note that the object must be opened by the main module later.

The function *eval_const* tries to evaluate an expression, which only succeeds if the expression contains only constants, i.e., no references to attributes and no shell commands.

The function *new_node* and *new_leaf* allocate space for a node in an expression tree and initialise it.

The function *process_handlers* is complicated by the fact that the different kinds of handlers that Gist uses must be converted to actions and translations in the format expected by the X Toolkit. The routine creates a translation table and installs it in the widget. One of the translations is for the mouse button that opens the editor for interactive editing of the object.

The conversion to translations and actions involves a few tricks. String messages are sent to widgets as ClientMessages, with a (decimally coded) pointer to the actual message. It is OK to pass pointers, since all widgets are within the same application. The handler for ClientMessages has a pointer to the list of handlers as first argument (as a decimal number) and the number of handlers as the second (again a decimal number).

Event handlers are implemented as actions for the event itself. (The event syntax should be checked, to shield the user from error messages coming from the X Toolkit, but currently that check is skipped.)

Finalize and Initialize handlers are implemented as actions for the «Unmap», respectively the «Map» event.

Mouse-down, click and double click handlers are just string handlers, because the mouse-down, click and double click events are synthetic events, generated by Gist itself (see the module «actions»)

Finally, *set_default_attribute* and *set_attribute* store the values of attributes in resources. The former stores the resource and value in the resource database, where it will act as a default value for all attributes of the same name. The latter uses *XtVaSetValues* to store the resource directly in a widget.

5.8 The «actions» module

The module is called «actions» both because it contains action functions in the terminology of the X Toolkit and because it executes the actions in Gist scripts.

The actions are registered with the X Toolkit with the help of a table, as follows:

```
XtActionsRec actions[] = {
{ "handle-string", handle_string },
{ "handle-event", handle_event },
{ "mouse-down", mouse_down },
{ "mouse-click", mouse_click },
{ "mouse-double", mouse_double },
{ "edit-script", edit_script },
```

```
};
```

The three mouse actions are used in the configuration file (see «configuration» below) to tell Gist when to generate the special mouse events. The *edit_script* action invokes the editor for interactively changing an object.

The *handle_string* action is called when the widget receives a message (a «ClientMessage»). All messages are handled by the same *handle_string* action. It searches through the list of message handlers for one that matches the messages and then calls a local function *execute* to execute the list of actions. The list of message handlers is the first parameter of the action function. It is put there by the *process_handlers* function in «parseaux» (see above).

The function *handle_event* handles raw events. There can be multiple *handle_event* actions in a single object, each bound to a different event. The first parameter of the function is a pointer to the handler to execute. Unlike *handle_string*, this action does not have to search through a list of handlers, neither does it have to check the conditions. All it has to do is call the *execute* function with the proper arguments.

5.9 The «classes» and «classes.def» modules

The «classes» module holds the information about the locally configured set of objects. It has routines to check for the existence of a certain object or a certain attribute. The «classes.def» file is a configuration file that is converted to a C file by a small program, *gconfig*. The result is a file with some arrays and functions that is included in the «classes.c» file.

Objects and attributes are not sorted and simple lineair search is used to find objects and attributes. The search is made a little faster because all names of objects and attributes are hashed so that only «quarks» have to be compared.

The main structure for holding information about available object classes is defined as follows:

```
typedef struct {
    String name;        /* The name of the class */
    XrmQuark q;         /* The name as a quark */
    WidgetClass *class_ptr; /* The type */
    ShellType tp;   /* MainWindow/DialogBox/NoShell */
    int nrattribs;      /* Number of resources */
    attrib_descr *attrib; /* List of resources */
    String def_trans;   /* Default translations */
} class_descr;
```

The class name of an object is not necessarily the same as the name of the widget used to implement it. Several objects may actually be implemented with the same widget, but with different attributes, different behaviour or different default values.

The same freedom exists in choosing names for attributes. Although the attributes of an object are actually the resources of the widget, they do not necessarily have the same name.

```
typedef struct {
    String attrib;   /* The name of the attribute */
    XrmQuark q;      /* The name as a quark */
    String resource; /* The name of the resource */
    Boolean set;     /* Can be set? */
    Boolean get;     /* Can be queried? */
} attrib_descr;
```

The booleans *set* and *get* indicate if an attribute can be assigned values (*set*) and if it can be used in an expression (*get*). Most attributes can be set (with *XtVaSetValues*), but only resources that have a converter from their own type to string can be queried (with *XtVaGetValues*), unless they are strings themselves, of course. It may be a good idea to add a few converters to Gist for common types (*int, float, Boolean*), since widget writers usually only provide converters in the other direction.

5.10 Configuration

As said elsewhere, every installation of Gist must define a set of objects, implemented with widgets. The configuration file «classes.def» defines the objects (their name, what ettributes they have, when they generate mouse events, etc.) and tells with what widgets they are implemented.

The format for configuration files is defined by the converter «gconfig», which is itself implemented in Awk. An example may serve to give the flavour of it, see figures 5.2 and 5.3.

For every object, Gist needs

- its name (CLASS),
- attributes (ATTRIB),
- if there is a converter from string to the type of each attribute,
- if there is a converter in the opposite direction, and
- if it is a window, box or a normal object (SHELL).

```
# The window object is actually a Board widget with a transientShell
# parent. It borrows all resources from board (which is not really a
# superclass, but the effect is the same.)
#
CLASS   window
SHELL   MainWindow
ID      xfwfBoardWidgetClass
FILE    Xfwf/Board.h
SUPER   board

# A box is also just a Board, but this time within an overrideShell.
#
CLASS   box
SHELL   DialogBox
ID      xfwfBoardWidgetClass
FILE    Xfwf/Board.h
SUPER   board

# A board is a general container for other objects. It has the resources
# of frame plus some that have to do with locations
#
CLASS   board
ID      xfwfBoardWidgetClass
FILE    Xfwf/Board.h
SUPER   frame
#
#       attribute       resource        set?    get?    default
#
ATTRIB  abs-x           XtNabs_x        y       y
ATTRIB  rel-x           XtNrel_x        y       y
ATTRIB  abs-y           XtNabs_y        y       y
ATTRIB  rel-y           XtNrel_y        y       y
ATTRIB  abs-width       XtNabs_width    y       y       20
ATTRIB  rel-width       XtNrel_width    y       y
ATTRIB  abs-height      XtNabs_height   y       y       20
ATTRIB  rel-height      XtNrel_height   y       y
ATTRIB  hunit           XtNhunit        y       y
ATTRIB  vunit           XtNvunit        y       y
ATTRIB  location        XtNlocation     y       y
```

Figure 5.2 A fragment of a configuration file for the installation of Gist.

The keywords ID, FILE, TRANS and CALLBACK...END provide the implementation of the object. They indicate the widget that is to be used, the header file to include, the trans-

```
# A button is a label with actions for mouse clicks. The actions draw
# the button's shadow in reverse when the mouse is pressed.
#
CLASS button
ID xfwfButtonWidgetClass
FILE Xfwf/Button.h
SUPER label
TRANS <Btn1Down>: set_shadow(sunken) mouse-down()
TRANS <Btn1Down>,<Btn1Up>: mouse-click() set_shadow()
TRANS <Btn1Down>(2+): mouse-double()

# Arrow
#
CLASS   arrow
ID      xfwfArrowWidgetClass
FILE    Xfwf/Arrow.h
SUPER   board
ATTRIB  direction       XtNdirection    y       n
ATTRIB  foreground      XtNforeground   y       n
ATTRIB  shadow          XtNarrowShadow  y       y
ATTRIB  initial-delay   XtNinitialDelay y       y
ATTRIB  repeat-delay    XtNrepeatDelay  y       y
#
# The callback is used to generate mouse-down events repeatedly
# as long as the button is pressed.
#
CALLBACK XtNcallback
{
    mouse_down(w, NULL, NULL, NULL);
}
END
```

Figure 5.3 Another fragment of a configuration file for the installation of Gist.

lations to install on the widget and any callbacks. The translations and callbacks should at least define when the objects generates mouse down, mouse click and mouse double click events. The example shows that the button widget generates mouse down on a «Btn1Down» event, whereas the arrow object generates it each time a certain callback is called.

The keyword SUPER is used to keep the descriptions short. In principle, each object should be accompanied by an exhaustive list of attributes. This list could become very long. The SUPER keyword says that all attributes of the named object

are inherited. Usually the object to inherit from is indeed the widget's superclass, but that need not be the case. For example, window «inherits» from board, since board has the same attributes as window, not because the widget that implements board is a superclass — it is, in fact, the *same* widget. —

From the list of attributes it is clear that not all attributes can be queried. This is caused by the lack of converter functions with string as the target type. *location* can be queried because it is a string itself. Numerical attributes can be queried, because Gist provides converters for some common data types. When widgets define new types and widget writers do not provide converters, it is impossible for Gist to translate the values back to strings.

Appendix A

Gist syntax

The syntax of Gist scripts is fairly simple. A script is basically a list of object definitions, each starting with the keyword «OBJECT». Each definition has two parts: the attributes and the handlers.

Before the first object, there may be a list of global attributes. They will act as default values for all objects that do not override them.

After the global attributes but still before the first object, there should also be a list of handlers that model the application for which this is an interface. The handlers can only be string handlers or regular expression handlers. They determine what actions must be executed when a certain string is outputted by the application. The actions usually distribute relevant portions of the output to the interface elements that can display that kind of information.

Below is the annotated syntax. All semantic actions have been left out, except for the parameters of the nonterminals. Instead there is a concise description of what each rule means and what semantics the parser extracts from it.

The syntax notation is close to that used by Yacc. Every rule has a head and a body, separated by a colon (:). The body ends with a semicolon (;). The body can contain alternatives, separated by vertical bars (|). For example, the rule

```
factor
  : IDENTIFIER
  | STRING
  | LPAR value RPAR
  ;
```

says that a *factor* can be formed in three ways: either from an IDENTIFIER, or from a STRING, or from an LPAR followed by a *value* followed by an *rpar*.

token	representation	token	representation
ACTIVATE	ACTIVATE	IDENTIFIER	*identifier*
ACTIVE	ACTIVE	INITIALIZE	INITIALIZE
BEEP	BEEP	LPAR	(
CLOSE	CLOSE	MINUS	-
CLOSED	CLOSED	MOUSECLICK	MOUSE-CLICK
COLON	:	MOUSEDOUBLE	MOUSE-DOUBLE
COMMA	,	MOUSEDOWN	MOUSE-DOWN
COMMAND	‘...command...‘	NEW	NEW
DEACTIVATE	DEACTIVATE	OBJECT	OBJECT
DIV	DIV	OPEN	OPEN
DOLLAR	$	PLUS	+
DOT	.	PRINT	PRINT
EVENT	{...*event*...}	REGEXP	/...*reg. exp*.../
FIELD	$*digit*	RPAR	)
FINALIZE	FINALIZE	STRING	"...*string*..."
HALT	HALT	TIMES	*

Figure A.1 *Terminals*

Parameters are added using the C notation, which leads to, e.g.,

```
factor(Expression *e)
  : IDENTIFIER
  | STRING
  | LPAR value(e) RPAR
  ;
```

A.1 Terminal symbols

The terminal symbols in the rules below are all written in uppercase. Most of them correspond closely to their representation in the script. Table A.1 lists them all.

A «command» is a normal shell command. An «identifier» is a sequence of letters, digits, underscores and dashes. An «event» is an event in the standard X Toolkit notation. A regular expression must be written in the POSIX extended syntax (but see section 4.8.2). The definition of the syntax can be found in UNIX manuals under regexp(5).

Note that SELF and PARENT do not appear as terminals. Although they have a clear semantic meaning that is different from other identifiers, that fact is not visible in the syntax.

A.2 Rules

The start symbol is *parse*. It is called either to create a new object or to set the attributes of an existing object. In the first case, the rule returns the widget it created, the type of object, and whether it should be opened during initialization. In the second case **widget* and **classname* already have a value. The parameter *def_parent* is both input and output, for if the created object is a window or box it will become the new default parent, otherwise the new object will be assigned the current value of *def_parent* if it does not specify an explicit parent.

```
parse(Widget *widgetp, XrmQuark *classname,
    Boolean *closed, Widget toplevel, GistObj *def_parent)
 : OBJECT class(classname, &class) unique_name(&name)
   parent_opt(&parent) closed_opt(closed)
   parse_attribs(*widgetp, *classname)
 | parse_attribs(*widgetp, *classname)
 ;
parent_opt(GistObj *parent)
 : parent(parent)
 | /* empty */
 ;
parent(GistObj *parent)
 : LPAR name(&name) RPAR
 ;
name(XrmQuark *name)
 : constant(name)
 ;
closed_opt(Boolean *closed)
 : CLOSED
 | /* empty */
 ;
class(XrmQuark *classp, WidgetClass *class)
 : IDENTIFIER
 ;
unique_name(XrmQuark *namep)
 : constant(namep)
 ;
```

A constant is an expression that can be evaluated at compile time by the parser itself. Such expressions are needed in a few places, such as the name of an object and the value of an attribute. The returned value is a string, hashed into an *XrmQuark*.

```
constant(XrmQuark *namep)
 : value(&expr)
```

```
  ;
```

Expressions are strings or numbers, connected by operators +, −, ∗ and DIV. For strings only + is meaningful. Strings and numbers are dynamically converted into each other when needed.

```
value(Expression *exprp)
 : term(exprp) more_terms(exprp)
 ;
more_terms(Expression *exprp)
 : plus_or_minus(&plus) term(&(*exprp)->node.right)
   more_terms(&(*exprp)->node.right)
 | /* empty */
 ;
plus_or_minus(Boolean *plus)
 : PLUS
 | MINUS
 ;
term(Expression *expr)
 : factor(expr) more_factors(expr)
 ;
more_factors(Expression *expr)
 : times_or_div(&times) factor(&(*expr)->node.right)
   more_factors(&(*expr)->node.right)
 | /* empty */
 ;
times_or_div(Boolean *times)
 : TIMES
 | DIV
 ;
factor(Expression *e)
 : IDENTIFIER attrib_opt(ident, e)
 | STRING attrib_opt(ident, e)
 | COMMAND attrib_opt(ident, e)
 | LPAR value(e) RPAR attrib_opt(ident, e)
 | FIELD attrib_opt(ident, e)
 ;
attrib_opt(XrmQuark ident, Expression *e)
 : DOLLAR IDENTIFIER
 | /* empty */
 ;
```

In an expression, an identifier on its own is just a string, but an identifier followed by a dollar sign and another identifier is a reference to an attribute. For example, `list` is the same as `"list"`, but `box1$list` is replaced by the current value of the *list* attribute of the object called *box1*.

A FIELD is a reference to the message that triggered a regular expression. It starts with a $. In an expression it will be replaced with the appropriate substring of the message that triggered the action.

The syntax of expressions is not yet completely satisfactory. For example, it is impossible to pass arguments to a shell command (COMMAND).

The definition of an object has two parts: the attributes and the handlers. The attributes are a set of keyword-value pairs: the name of the attribute followed by a constant expression that gives the value. The expression must be constant, because the value will be installed in the widget (with *XtVaSetValues*) before the whole script has been read.

```
parse_attribs(Widget widget, XrmQuark class)
 : attributes(widget, class)
   handlers(class, &handlers, &n)
 ;
attributes(Widget widget, XrmQuark class)
 : attribute(widget, class) attributes(widget, class)
 | /* empty */
 ;
handlers(XrmQuark class, Handler **handlers, int *n)
 : handler(class, &(*handlers)[*n])
   handlers(class, handlers, n)
 | /* empty */
 ;
attribute(Widget widget, XrmQuark class)
 : resource(class, &resource) constant(&val)
 ;
resource(XrmQuark class, XrmQuark *resource)
 : IDENTIFIER
 ;
```

There are four types of handlers. *string_handler*s and *regexp_handler*s are triggered by text messages, either coming from other objects or — in the case of the special input widget — from the application. *event_handler*s provide the means to attach actions to general events. The *special_handler*s are handlers for the mouse-down, mouse-click and mouse-double-click events that are generated by Gist itself and for the «map» and «unmap» events that occur when an object is opened or closed.

```
handler(XrmQuark class, Handler *h)
 : string_handler(class, h)
 | regexp_handler(class, h)
```

```
  | event_handler(class, h)
  | special_handler(class, h)
  ;
string_handler(XrmQuark class, Handler *handler)
  : STRING
    condition_opt(handler) COLON
    actionlist(class, &handler->actions) DOT
  ;
regexp_handler(XrmQuark class, Handler *handler)
  : REGEXP
    condition_opt(handler) COLON
    actionlist(class, &handler->actions) DOT
  ;
```

It is not possible to attach conditions to event handlers, because it is impossible to install two translations for the same event. This is an unfortunate restriction in Gist, but it would be very expensive to fix. The special events (below) *can* have conditions, however.

```
event_handler(XrmQuark class, Handler *handler)
  : EVENT COLON actionlist(class, &handler->actions) DOT
  ;
special_handler(XrmQuark class, Handler *handler)
  : special_event(&handler->tp) condition_opt(handler)
    COLON actionlist(class, &handler->actions) DOT
  ;
special_event(HandlerType *tp)
  : FINALIZE
  | INITIALIZE
  | MOUSEDOWN
  | MOUSECLICK
  | MOUSEDOUBLE
  ;
condition_opt(Handler *handler)
  : condition(handler)
  |
  ;
condition(Handler *handler)
  : plus_or_minus(&plus) closed_or_sensitive(&closed)
  ;
closed_or_sensitive(Boolean *closed)
  : CLOSED
  | ACTIVE
  ;
```

The comma between actions is optional. It can sometimes help to improve the lay-out, especially when actions are typed

one after another on the same line.

```
actionlist(XrmQuark class, Action *actions)
 : action(class, *actions)
   more_actions(class, &(*actions)->next)
 |
 ;
more_actions(XrmQuark class, Action *actions)
 : COMMA action(class, *actions)
   more_actions(class, &(*actions)->next)
 | action(class, *actions)
   more_actions(class, &(*actions)->next)
 |
 ;
```

The different kinds of actions are clearly recognizable in the following rule. The first alternative refers to the *ASend* action, that sends a message to another object. It is also used to set attributes in the receiver. In this case the value for the attribute need not be a constant, since it can be evaluated at run-time.

The second alternative creates a new object, by cloning the named object and all its children. The actions *AOpen* and *AClose* open and close objects, respectively. The *AActivate* and *ADeactivate* actions make objects (in)sensitive to user input. The *APrint* action sends a string to the application, which will appear on the application's standard input.

AHalt simply stops execution of both Gist and the application. It is only used in exceptional cases, since normally the application determines when the interface should stop, instead of the other way round. The *ABeep* action sounds the terminal bell.

```
action(XrmQuark cl, Action act)
 : receiver(&act->receiver)
   resource_opt(cl, &act->sub) value(&act->value)
 | CLONE receiver(&act->receiver) AS value(&act->receiver)
 | OPEN receiver(&act->receiver)
 | CLOSE receiver(&act->receiver)
 | ACTIVATE receiver(&act->receiver)
 | DEACTIVATE receiver(&act->receiver)
 | PRINT value(&act->value)
 | HALT
 | BEEP
 ;
resource_opt(XrmQuark class, Expression *sub)
 : DOLLAR resource(class, &attrib)
 |
```

```
 ;
parent_expr_opt(Expression *sub)
 : parent_expr(sub)
 |
 ;
receiver(Expression *receiver)
 : value(receiver)
 ;
```

Appendix B

Gist example

This example contains a complete interface to the game of tic-tac-toe — or crosses and noughts — as found on many UNIX machines. The game was chosen for the simplicity of its I/O. Nevertheless, the example shows all the features of Gist. A larger example would simply be «more of the same».

We assume most people now the rules of the game: a 3×3 square must be filled by two players, taking turns. One player draws X's, the other O's. The player that gets three symbols in a row wins. In the computer version, the computer plays with O's and the user with the X's.

The output of the tic-tac-toe program consists of the following texts:

`Accumulated knowledge?`
: This is asked at the start of the game and must be answered with y or n. When answered with y, the program will learn from its mistakes.

```
123
456
789
```
: This is actually three lines, so the interface has to collect them before actions can be taken. The three lines contain the digits 1 to 9, with X's and O's replacing the digits if the corresponding position has been crossed by either the computer or the user/player.

`You win`
: The user wins.

`I win`
: The program wins.

`Your move?`
: When the user has to choose a square.

`Thinking`
When the computer is choosing a square.

The input is even simpler. At the start of the game the user must enter y or n in response to the question about learning. In the course of the game the user must enter digits between 1 and 9 and at the end of the game again y or n in answer to the question about starting another game.

The interface will be modeled as a direct manipulation interface. As much as possible the user is given control over the dialogue. The playing field is shown on the screen and the user can click on empty squares. The question about learning right after startup is shown in a dialog box. There is a quit-button that the user can click at any time to stop the game. Error messages are not needed, since the only thing the user can do wrong – clicking on a square that is already marked – can be easily intercepted by the interface itself.

«You win» and «I win» messages are displayed in another dialog box, that has just an OK button.

The interface will therefore consist of a main window with ten buttons (nine squares plus quit), a dialog box with a yes and a no button, and a dialog box with an OK button.

B.1 Application model

The script starts with the input model. There are handlers for each of the possible outputs of the application. The question «Accumulated knowledge?» results in the opening of the modal dialog box called *learn*. Lines with just X's, O's and digits are catched by a regular expression and are appended to a local variable, until the text «...your turn...» is received. At that moment the stored text is taken apart by a separate handler and used to set the labels of the nine buttons. The nine parenthesised parts of the regular expression correspond to the nine fields $1 to $9 in the action list.

```
"Accumulated knowledge?": OPEN learn.
"You win"               : show-win $0.
"I win"                 : show-win $0.
/[XO1-9]+/              : SELF$state  "" + SELF$state + $0.
"Thinking"              : SELF  "do " + SELF$state.
"Your move?"            : SELF  "do " + SELF$state.
/do (.)(.)(.)(.)(.)(.)(.)(.)(.)/:
        "1" $1,  "2" $2,  "3" $3
        "4" $4,  "5" $5,  "6" $6
        "7" $7,  "8" $8,  "9" $9.
```

SELF is a predefined name, which always refers to the current object, even if that object is the nameless input object. The

first line opens the *learn* box (defined below). The second line sends the message «You win» to the *show-win* object. The `$0` represents the message that triggered the action, which is in this case simply «You win».

A «virtual attribute» *state* is used to collect the three lines that represent the board. The fourth handler is triggered by any input that contains just three letters, this will match the three lines that represent the board. *state* will be the concatenation of these three lines. The addition of the empty string at the beginning is there to force the + to be interpreted as string concatenation and not as addition of numbers.

B.2 Dialog boxes

The dialog box *learn* displays a text and two buttons. It is initially closed, since it will be opened by the input object above.

```
OBJECT box learn CLOSED
  width   300
  height  200

OBJECT label label
  location  "0 0 1.0 1.0-40"
  label     "Should the program learn from mistakes?"

OBJECT button yes
  location  "0.5-100 1.0-30 80 20"
  label     "Yes"
  MOUSE-CLICK: PRINT "y\n", PARENT CLOSE.

OBJECT button no
  location  "0.5+20 1.0-30 80 20"
  label     "No"
  MOUSE-CLICK: PRINT "n\n", PARENT CLOSE.
```

Note that the identifier *label* is used in three different roles: type of object, name of object and name of attribute. Gist is able to keep the three usages separate, because they always occur in specific contexts.

The quotes around Yes and No are optional. When a value does not include whitespace, the quotes can be omitted.

PARENT is a predefined identifier, which always refers to the parent of the current object. In this case it refers to the *learn* box.

The second dialog box is called *show-win*. It displays either the text «You win» or «I win», but the way it is defined below it just displays any text that is sent to it. It has a single

button that pops down the box. The text to display is sent as a message to the box and the box asigns it to the label, and then opens itself.

```
OBJECT box show-win CLOSED
  width   300
  height  200
  /.*/:           # matches any message
        win-label$label $0
        SELF OPEN.

OBJECT label win-label
  location  "0 0 1.0 1.0-40"

OBJECT button ok
  location  "1.0-40 1.0-30 80 20"
  label     "OK"
  MOUSE-CLICK: PARENT CLOSE.
```

B.3 Main window

The main window is given a title and a size. When it is opened, it creates the nine buttons for the nine squares and assigns them a position. Since there are only nine buttons, there is no real need for creating them dynamically, except as an example.

```
OBJECT window tic-tac-toe
  width   80
  height  120
  title   tic-tac-toe

  initialize:
    CLONE proto-btn AS 1,  1$x 10,  1$y 10,
    CLONE proto-btn AS 2,  2$x 30,  2$y 10,
    CLONE proto-btn AS 3,  3$x 50,  3$y 10,
    CLONE proto-btn AS 4,  4$x 10,  4$y 30,
    CLONE proto-btn AS 5,  5$x 30,  5$y 30,
    CLONE proto-btn AS 6,  6$x 50,  6$y 30,
    CLONE proto-btn AS 7,  7$x 10,  7$y 50,
    CLONE proto-btn AS 8,  8$x 30,  8$y 50,
    CLONE proto-btn AS 9,  9$x 50,  9$y 50.
```

The nine buttons that form the 3×3 square each have a handler for mouse clicks and two handlers that are triggered by the message from the input objects. Note that that message can be either an X, an O or a digit. There are two handlers, because the X and O are treated differently from the digit. When the digit is received, nothing will be displayed, if the X

or O is received, it will be displayed.

```
OBJECT button proto-btn
  width 20
  height 20
  MOUSE-CLICK: PRINT SELF$label + "\n".
  /[XO]/:      SELF$label $0.
  /[1-9]/:     SELF$label "".
```

The last object in the interface is the quit button. It simply aborts the program when it is clicked.

```
OBJECT button quit
  location  "10 80 60 15"
  label     "Quit"
  MOUSE-CLICK: SELF HALT.
```

B.4 Flagging impossible moves

It was said in the introduction to this chapter that the interface could prevent erroneous moves, but the definitions above do not yet do that. The obvious place to check for such moves is in the buttons, since the buttons contain the necessary information – an empty or non-empty label – and also pass the move on to the application.

Each of the nine buttons above should therefore be changed as follows: the handler for mouse clicks must be replaced by the two handlers

```
MOUSE-CLICK: SELF  "?" + SELF$label.
"?": PRINT "1\n".
```

with the 1 replaced by the appropriate digit for that button. The button now sends a message to itself consisting of a question mark concatenated with the current label. Only if the label is empty will the message match the next handler and cause a digit to be sent to the application. If the message turns out to be «?X» or «?O» it will simply be ignored, since there is no handler for it.

But the ignored message can also be used to give a warning signal to the user, for example in the form of a short beep. That can be accomplished by adding the next handler to all nine buttons:

```
/?./: SELF BEEP.
```

B.5 Using higher level objects

The interface contains two dialog boxes, that have in this case been constructed from a box, a label and one or more buttons.

Such boxes are so common that there is likely to be a special, integrated object for this task. If we assume an object type *alert*, we can replace the two boxes with two alerts. The *learn* box becomes:

```
OBJECT alert learn
  message  "Should the program learn from mistakes?"
  buttons  "Yes|No"
  "1": PRINT "y\n".
  "2": PRINT "n\n".
```

The alert object has attributes *message* and *buttons*. The latter contains the labels for the buttons, separated by |. The alert object is further assumed to close itself when a button is clicked and send the number of the pressed button – 1 or 2 – to itself.

The *show-win* box can become:

```
OBJECT alert show-win
  buttons  "OK"
  /..+/: SELF$message $0, SELF OPEN.
```

A slight subtlety is involved in the regular expression. We want it to match the texts «You win» and «I win», but not the digit «1» that the alert sends to itself when the OK button is pressed. A simple solution is to require the message to contain at least two characters.

Bibliography

Apple Computer Inc. (1987). *Human interface guidelines: the Apple desktop interface*. Addison Wesley, Reading, MA. Style guide.

Apple Computer Inc. (1988). *Hypercard script language: the Hypertalk language*. Addison-Wesley, Reading, MA.

G. Avrahami, K. Brooks and M. Brown (1989). 'A two-view approach to constructing interfaces'. In: SIGGRAPH *'89 Conference Proceedings*, pages 137–146. as cited in Heeman [1992].

Simon Been (1993). 'E-mail: en dan nu het (slechte) nieuws'. In: *Computable*, vol. 26 nr. 16, pages 17–19, April 1993. (in Dutch).

Maurice Bergmans (1993). 'X, osf/motif en x-designer, een technisch overzicht'. In: *Werken met Sun*, February 1993.

Jan van den Bos (1988). 'Abstract interaction tools: a language for user interface management systems'. In: *ACM Transactions on Programming Languages and Systems*, vol. 10 nr. 2, pages 215–247, April 1988.

Jan van den Bos and Chris Laffra (1990). 'Project DIGIS, building interactive applications by direct manipulation'. In: *Computer Graphics Forum*, nr. 9, pages 181–193, 1990.

C. Marlin 'Lin' Brown (1988). *Human-computer interface design guidelines*. Ablex publishing corp., Norwood, New Yersey.

Marc H. Brown and James R. Meehan (1992). *The Forms*VBT *reference manual*. A draft version. Comments are most welcome. (Available by ftp.).

D. Browne, M. Norman and D. Riches (1990). 'Why build adaptive systems?'. In: **Dermot Browne, Peter Totterdell and Mike Norman** (eds.), *Adaptive user interfaces*, chapter 1, pages 15–58. Academic Press / Harcourt Brace Jovanovich, London, 1990. Computers and people series.

D. Browne, P. Totterdell and M. Norman (eds.) (1990). *Adaptive user interfaces*. Computers and people series. Academic Press, London.

Hans de Bruin, Peter Bouwman and Jan van den Bos (1993). 'DIGIS, a graphical user interface design environment for non-programmers'. In: *Proceedings Eurographics '93, Barcelona*, Amsterdam, 1993. North-Holland. To be published.

Mark A. Clarkson (1991). 'An easier interface'. In: *Byte*, vol. 16 nr. 2, pages 277–282, February 1991.

Pierpaolo Degano and Erik Sandewall (eds.) (1983). *Integrated interactive computing systems*. North Holland.

Edsger W. Dijkstra (1987). 'The humble programmer'. In: *ACM Turing award lectures, the first twenty years 1966–1985*. ACM Press/Addison Wesley, Reading, Mass., 1987.

James D. Foley (1987). 'Interfaces for advanced computing'. In: *Scientific American*, vol. 257 nr. 4, pages 83–90, October 1987.

Adele Goldberg (1983). *Smalltalk-80: the language and its implementation*. Addison-Wesley, Reading, Mass.

Charles F. Goldfarb (1991). *The* SGML *handbook*. Oxford University Press, New York.

Nick Hammond, Margaret M. Gardiner, Bruce Christie and Chris Marshall (1987). 'The role of cognitive psychology in user-interface design'. In: **Margaret M. Gardiner and Bruce Christie** (eds.), *Applying cognitive psychology to user-interface design*, pages 13–53. John Wiley & Sons, Chichester, 1987.

Frans C. Heeman (1992). *State-of-the-art in window systems and* UIMS*'s*. Report 92/07, Software Engineering Research Center (SERC), Utrecht.

D. Austin Henderson Jr. (1986). 'The trillium user interface design environment'. In: **Marilyn Mantei and Peter Orbeton** (eds.), *Human factors in computing systems III*, pages 221–227. North-Holland, Amsterdam, 1986.

Richard Hesketh (1992). *The Dirt User Interface Builder*. Technical report, Computing Laboratory, University of Kent, Canterbury.

H. J. A. Hofland (1992). 'Aan de andere kant van de spiegel'. In: *NRC Handelsblad (Cultureel Supplement)*, nr. 1133, pages 1–2, August 1992. (Newspaper 14-8-92, in Dutch).

Mark R. Horton (1983). *Standard for interchange of* USENET *articles*. RFC 850, USENET.

Edwin L. Hutchins, James D. Hollan and Donald A. Norman (1986). 'Direct manipulation interfaces'. In: **Stephen W. Draper** (ed.), *User centered system design*, chapter 5, pages 87–124. Lawrence Erlbaum, Hillsdale, NJ, 1986.

IBM (1989). *Common User Access, advanced interface design guide*. Technical Report SC26-4582-0, IBM. Style guide.

Bob Jacobson (1992). 'The ultimate user interface'. In: *Byte*, vol. 17 nr. 4, pages 175–182, April 1992.

S. C. Johnson (1975). *Yacc – yet another compiler compiler*. Computing science technical report 32, AT & T Bell Laboratories, Marray Hill, N.J.

Richard D. Lasky (1989). 'Hypertalk program design'. In: *Byte*, vol. 14 nr. 8, pages 205–210, August 1989.

M. E. Lesk (1975). *Lex – a lexical analyzer generator*. Computer science technical report 39, Bell Laboratories, Murray Hill, N.J.

M. Linton, J. Vlissides and P. Calder (1989). 'Composing user interfaces with interviews'. In: *IEEE Computer*, vol. 20 nr. 1, pages 32–44, 1989.

Paul Luff, Nigel Gilbert and David Frohlich (eds.) (1990). *Computers and conversation*. Computers and people series. Academic Press, London.

Lindsay MacDonald (1991). 'Smart use of color in displays'. In: *Byte*, vol. 16 nr. 13, pages 84IS.35–84IS.46, December 1991.

Chris Marshall, Bruce Christie and Margaret M. Gardiner (1987). 'Assessment of trends in the technology and techniques of human-computer interaction'. In: **Margaret M. Gardiner and Bruce Christie** (eds.), *Applying cognitive psychology to user-interface design*, pages 279–312. John Wiley & Sons, Chichester, 1987.

Microsoft Corporation (1991). *The* GUI *guide, localizing the graphical user interface.*

Paul Molenaar (1988). *Het HyperCard handboek.* Addison-Wesley, Amsterdam.

C. Musciano (1988). *Tooltool user's guide, version 2.1.* Technical report, Advanced Technology Department, Harris Corporation.

Donald A. Norman (1988). *The psychology of everyday things.* Basic Books, New York.

Donald A. Norman and Stephen W. Draper (1986). *User centered system design: new perspectives on human-computer interaction.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Open Software Foundation (1991). *OSF/Motif Style Guide.* Prentice Hall, Englewood Cliffs, New Jersey, revision 1.1 edition.

John K. Ousterhout (1993). *An introduction to Tcl and Tk.* Addison-Wesley. This is a partial draft of a book to be published in 1993. The draft version is available by anonymous FTP.

Randy Pausch, Nathaniel R. Young II and Robert DeLine (1991). 'SUIT: the pascal of user interface toolkits'. In: *Proc. of the* ACM *symposium on user interface software and technology*, pages 117–125.

Jon Peddie (1992). *Graphical user interfaces and graphic standards.* McGraw-Hill,, New York.

M. Rochkind (1989). 'Xvt: a virtual toolkit for portability between window systems'. In: *Proceedings of the winter 1989* USENIX *conference*, pages 151–163.

Frank Rose (1989). *West of Eden, the end of innocence at Apple Computer.* Penguin, Harmondsworth, Middlesex, England.

Yen-Ping Shan (1991). 'An object-oriented framework for direct-manipulation user interfaces'. In: **E. H. Blake and P. Wisskirchen** (eds.), *Advances in object-oriented graphics I*, chapter 1, pages 3–19. Springer, Berlin, 1991. (Eurographic Seminars) Contains extensively revised versions of some of the papers presented at the Eurographics workshop on object-oriented graphics, held in Königswinter, Germany in June 1990.

K. Shmucker (1986). 'Macapp: an application framework'. In: *Byte*, vol. 11 nr. 8, pages 189–193, August 1986.

Ben Shneiderman (1987). *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley, Reading, Mass. Reprinted May, 1987.

G. Singh, C. Kok and T. Ngan (1990). 'Druid: a system for demonstrational rapid user interface development'. In: *Proceedings of the* ACM SIGGRAPH *symposium on user interface software and technology (*UIST *'90)*, pages 167–177, Snowbird, Utah, 1990.

D. Smyth (1991). 'Wcl – widget creation library, a thin veneer over xrm'. In: *5th annual X technical conference*, Boston, 1991.

Lee Sproull and Sara Kiesler (1991). 'Computers, networks and work'. In: *Scientific American*, vol. 265 nr. 3, pages 84–91, September 1991.

Joseph W. Sullivan and Sherman W. Tyler (eds.) (1991). *Intelligent user interfaces*. ACM Press frontier series. ACM Press / Addison-Wesley, Reading, Massachusetts.

Sun Microsystems (1990). *Open Look graphical user interface technical reference*, volume 1 & 2. Addison Wesley, Reading, Massachusetts. Contains style guide.

Harold Thimbleby (1990). *User interface design*. ACM Press Frontier series. Addison-Wesley, Wokingham, England.

Pietje van der Velden (1992). *Analyzing design tasks and concepts in user-interface design*. report 92/05, Software Engineering Research Center (SERC), Utrecht.

B. Webster (1989). *The NeXT book*. Addison Wesley, Reading, Massachusetts.

Charles M. M. de Weert (1988). 'The use of color in visual displays'. In: **Gerrit C. van der Veer and Gijsbertus Mulder** (eds.), *Human-computer interaction: psychonomic aspects*, pages 26–40. Springer Verlag, Berlin, 1988.

G. M. Welling (1991). 'Van dienstbodes, databases en didactiek: "al doende leert men"'. In: *VGI-cahiers 4*, pages 78–96. Hilversum, 1991.

George M. Welling (1992). 'Intelligent large-scale historical direct-data-entry programming'. In: *History & Computing V, Proc. 5th int. congress AHC 1990*, pages 563–571, Montpellier, September 1992.

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

Robin Wooffitt (1990). 'On the analysis of interaction'. In: **Paul Luff, Nigel Gilbert and David Frohlich** (eds.), *Computers and conversation*, chapter 1, pages 7–38. Academic Press / Harcourt Brace Jovanovich, London, 1990. Computers and people series.

Samenvatting

Het gedeelte van een computerprogramma dat zich bezig houdt met de communicatie tussen programma en gebruiker is het gebruikersinterface. Programma's kunnen zeer ingewikkeld worden of ze kunnen gebruikt moeten worden door onervaren gebruikers. Het belangrijkste middel om de computer dan toegankelijker te maken is een aantrekkelijk interface. Voor veel toepassingen betekent dat gebruik van de muis en grafische elementen op het scherm, voor sommige is een natuurlijke taal (bijvoorbeeld communiceren in Nederlands of Engels) de aangewezen weg. Bij het maken van zo'n interface doen zich een aantal problemen voor, zoals:

- Het is voor de programmeur veel werk om een interface te implementeren. De talen waarin applicaties worden geschreven zijn vaak niet zo geschikt voor het schrijven van grafische interfaces. Meestal wordt een programma meer dan tweemaal zo lang als een grafisch interface wordt toegevoegd.
- Het is niet te voorzien hoe een interface uiteindelijk voldoet. Maar prototypes maken is moeilijk, door het vele programmeerwerk. Toch zou het eigenlijk nodig zijn, omdat het nu eenmaal niet mogelijk is een formele specificatie te geven van een interface. Mensen zijn daarvoor te complex.
- Aanpassen van een interface in een later stadium wegens overbrengen naar een andere taal of voldoen aan specifieke wensen van de gebruiker kan vaak alleen met behulp van de oorspronkelijke programmatekst (en soms alleen door de auteur...). Het inbouwen van mogelijkheden voor de gebruiker om zelf dingen te wijzigen kost extra inspanning.

- Interfaces zijn vaak gebonden aan een bepaalde computer, ook al zou de applicatie zelf eenvoudig op andere typen computers kunnen draaien.

Voor deze problemen zijn oplossingen gezocht in verschillende richtingen, de meeste in de vorm van hulpmiddelen voor de programmeur.

Een interface voldoet als de gebruiker er met plezier mee werkt (subjectief) en als hij er sneller mee kan werken dan met andere programma's (objectief). Vooraf is het resultaat nauwelijks te garanderen, hoewel er inmiddels wel zoveel inzicht is verkregen, dat er enkele algemene richtlijnen zijn te geven.

Het blijkt echter dat veel programmeurs geen idee hebben van de eigenschappen van een goed interface. Deze dissertatie probeert een beeld te geven van interfaces, met name van grafische interfaces. Daarbij komen inzichten uit informatica, sociologie, psychologie en ergonomie bij elkaar.

Het resultaat is een voorstel voor een benadering die weliswaar niet voor alle gevallen kan voldoen, maar die wel kan helpen bij het grootste deel van de toepassingen zoals die nu worden gemaakt. De methode is gebaseerd op twee principes: modulariteit en «het juiste gereedschap».

Modulariteit, omdat het interface zodanig anders van karakter is dan het achterliggende programma, dat het het beste is voor beide delen eigen ontwikkelmethoden te gebruiken. Beide delen zijn bovendien gebaat bij een eigen programmeertaal.

De aangewezen methode voor het interface is *prototyping* of *iterative design*, terwijl voor de applicatie *formele specificatie* beter werkt. De taal voor de applicatie is meestal een traditionele programmeertaal, het interface vraagt om een speciale taal, bijvoorbeeld de door voor dit onderzoek ontwikkelde Gist.

Relatief nieuw in deze benadering is (de uitwerking van) het idee dat een interface door een ontwerper gemaakt moet kunnen worden, zonder tussenkomst van een programmeur.

Het hoofdstuk «Human factors & GUI's» beschrijft hoe grafische interfaces worden ontwikkeld en aan welke eisen ze moeten voldoen. Het belangrijkste concept in dit verband is *direct manipulation*. In deze theorie wordt ervan uitgegaan dat veel dingen kunnen worden gerepresenteerd door een plaatje of symbool en dat acties heel goed duidelijk kunnen worden gemaakt door het plaatje van vorm of van plaats te laten veranderen. Dat geldt zelfs voor abstracte objecten

als files en directories. Het kopiëren van een file naar een andere directory kan bijvoorbeeld worden uitgebeeld door het plaatje dat de file voorstelt uit de ene rechthoek op het scherm naar de andere te bewegen. Met behulp van de muis kan de gebruiker dit ook zelf doen.

Het hoofdstuk «User interface development systems» beschrijft de meest gebruikte elementen in grafische interfaces, zoals *windows, butttons* en *icons*. Het zijn de elementen die typerend zijn voor direct manipulation, hoewel ze niet alleen daarvoor gebruikt hoeven te worden.

Deze dissertatie beschrijft een software-pakket dat in het kader van het onderzoek is ontwikkeld. Gist is bedoeld voor het ontwerpen en implementeren van grafische interfaces. Het is een UIDS met een geïnterpreteerde, «object-oriented» taal voor het beschrijven van interfaces. De sterke punten van Gist zijn de volgende:

- Gist is eenvoudig te gebruiken. Het kan gebruikt worden als interface voor programma's in elke programmeertaal, zelfs voor shell-scripts. Dit is vergelijkbaar met Tool-Tool,[1] maar de interfaces die daarmee te maken zijn, zijn zeer beperkt.
- Doordat invoer beschreven wordt met *reguliere expressies* is het eenvoudig Gist aan te sluiten op allerlei bestaande programma's.
- Gist is *object-oriented,* zodat het beter aansluit bij het intuïtieve model dat programmeur en gebruiker van een interface hebben.
- Alle objecten hebben *defaults* die voor veel toepassingen voldoende zijn. De defaults kunnen achteraf nog worden vervangen.
- Gist is een *taal.* Dat maakt het mogelijk het interface op papier af te drukken en op te nemen in de documentatie en in versie-controle programmatuur, zoals SCCS en RCS.
- Omdat Gist geïnterpreteerd wordt, kan het interface nog veranderd worden als het applicatieprogramma al klaar is, dus ook door de gebruiker zelf.
- De interpreter werkt ook los van de applicatie en kan gebruikt worden om interfaces te testen en prototypes te laten zien.
- Gist kan worden geconfigureerd met willekeurige interface elementen en is dus niet gebonden aan bijvoorbeeld Motif of Athena widget sets.

[1] zie Musciano [1988]

Gist heeft ook nadelen:

- Gist helpt niet bij het voldoen aan «guidelines» of vuistregels voor het creëren van interfaces. M.a.w. het voorziet in alle noodzakelijke materiaal maar helpt niet bij het correct gebruik ervan.
- Er is geen *direct manipulation* editor voor het creëren van interfaces.

Index