Designing Graphical Interface Programming Languages for the End User

Gary Marsden

January 1998

Contents

List of Figures

Abstract

This thesis sets out to answer three simple questions: What tools are available for novice programmers to program GUIs? Are those tools fulfilling their role? Can anything be done to make better tools? Despite being simple questions, the answers are not so easily constructed.

In answering the first question, it was necessary to examine the range of tools available and decide upon criteria which could be used to identify tools aimed specifically at the novice programmer (there being no currently agreed criteria for their identification). Having identified these tools, it was then necessary to construct a framework within which they could be sensibly compared.

The answering of the second question required an investigation of what were the successful features of current tools and which features were less successful. Success or failure of given features was determined by research in both programming language design and studies of programmer satisfaction.

Having discovered what should be retained and discarded from current systems, the answering of the third question required the construction of new systems through blending elements from visual languages, program editors and fourth generation languages. These final prototypes illustrate a new way of thinking about and constructing the next generation of GUI programming languages for the novice.

Acknowledgements

Firstly I must thank my initial supervisor, who has become both a mentor and a friend. It would be impossible to quantify the effect Harold has had on my life (most of it positive I hasten to add.) Thanks Harold, you are a unique person and it is a privilege to know you.

Thanks are also due to Richard Bland (my replacement supervisor) who has not only managed to protect me from the ravages of Stirling beauraucracy, but has done much to encourage me in my lecturing career and to keep me in gainful employment.

Thanks to Al (my replacement, replacement supervisor!) for his view of "real" computer science and an outstanding choice of car.

I am also indebted to DENI who funded this work for two years; to the Department of Computer Science at Stirling who employed me subsequently; to the School of Computing Science at Middlesex University who currently employ me and have generously given me the time to finish this work.

This thesis has taken me five years on and off to complete. During that period *a lot* of people have encouraged and discouraged me. To the Stirling Ph.D. discouragers (Geoff, Paul, Andy, Flash and Steve), yes I should have listened. To the chief discouragers Dave and Kev, thanks guys for all the distractions over the years and our various get rich slow schemes.

To the encouragers at Stirling (Anne, Kath, Aoife, Ana) and those at Middlesex (Matt and Ann especially); thanks, it is good to know that someone thinks I am capable of finishing this.

To my parents; I deeply appreciate the support they have given me over the years, no matter how insane what I was doing may have seemed to them. To my dad, one of the most intelligent men I know, who showed me that "footering" with technology is perfectly good and respectable way to earn a living. To my mum, who persevered for years with my creative spelling, imaginative grammar and encrypted handwriting; she is responsible for any coherent

structure this document might contain. I appreciate your faith in me and it looks like I am going to finish after all.

Finally to Gil, who very bravely married me and my unfinished thesis. She has been with this *thing* right from the studentship interviews through to the final proof reading. She has supported me physically, emotionally and spiritually; more than I could ever reasonably expect.

This thesis is dedicated to her.

Preface

Before reading the main body of the text, it is best to read this preface as it will help to explain why this thesis was written and why it has turned out the way it has. The preface also provides a chance to summarise my ideas, so that you have some idea of what to expect.

To begin with, the motivation for the thesis came from trying to implement applications for an undergraduate dissertation [Mar92] using HyperCard. This resulted in the conclusion that there *had* to be a better way of doing things. Having discovered that there wasn't, I felt that something needed to be done. Whether or not this thesis has *done* what it should have is not certain; at the very least, however, something has been started to have been done.

Work programme

I initially thought that my programme of work would consist of writing a new system then discussing it and telling you how wonderful it was. When conducting my literature review, however, it became apparent that the last thing the computer science community needed was yet another prototype system. The number of experimental, research and commercial languages that are available to prospective programmers is growing continuously; most driven rapidly by new technologies, but having little theoretical grounding. Instead of adding to the mess, I thought that I would use the opportunity of a Ph.D. to review as many tools as I could find and try to rationalise and analyse the wider field. Having identified what was wrong, I hoped to use the failures (and occasional successes) in current systems to suggest and develop better ways of creating new systems.

Analysing current systems

To start this work, I had to examine and use current graphical interface programming systems, such as HyperCard and Visual Basic. To test these systems fully and expose their strengths and weaknesses, I undertook several large programming tasks. Examples of these tasks are as follows:

- To test Visual Basic, I created a graphical interface to a neural network application. This revealed many flaws in the design and implementation of Visual Basic, but the program itself (combined with the neural network application) won the John Logie Baird award for innovation (1995) and secured a large development grant from the Scottish Office.

- To test AuthorWare and AppleScript, I joined a UK wide CAL project for a month and implemented various teaching modules for the "Business Mathematics" module. Due to the limitations in these pieces of software, this project was eventually abandoned after a total of two years work.

Another way in which these systems were investigated was their use in implementing the prototypes used in the developing the ideas presented in subsequent chapters. This produces a slightly recursive investigation — i.e. I was implementing solutions to the problems of one system in a second system; itself having problems which would require the implementation of a solution! HyperCard, Clean, Java and tcl/tk were all used in this way.

Output

Extracts from this work have been published to provide feedback as the work developed. Refereed output consists of the following documents: [Mar94], [Mar95] and [MC97]. Furthemore, some of the ideas for providing end user customisation (discussed in chapter five) are now being used in a commercial software package [Inn97].

Motivation and frustration

Throughout the period of completing this Ph.D. I have been required to do a lot of teaching, which has proven to be a mixed blessing. Despite the obvious drawbacks of time commitment, teaching novice programmers languages like Pascal, SQL, Java and Gofer has allowed me to witness the types of confusions they experience.

This has been highly motivating as I believe that the needs of the novice programmer are largely unmet by new language developments in either industry or academia. Evidence for this belief is given in the first chapter, but in summary:

- The commercial sector are producing languages which people want to use — languages which facilitate the creation of highly graphical applications. However, commercial developers have ignored a lot of the computing science research available on the design and implementation of programming languages. Commercial languages seem to be developed by continually adding features, without regard for the complexity this inevitably introduces.

- It seems that systems such as HyperCard and Visual Basic do not fit comfortably into any single discipline. They are not quite visual languages, not quite full UIMSs and could not be considered a "proper" language by those interested in programming language design. Yet, these are the languages which are accessible to the greatest number of people.

As a consequence, this thesis is not about rushing into the technological future and creating an Internet aware, distributed end user programming language (the newest technologies at the time of writing) — the rest of the computer industry is already doing this. Instead, I want to take stock of where we are, take a look at what has gone before and demonstrate principles for creating future generations of tools.

Chapter 1

Introduction

"If a man will begin with certainties, he shall end in doubts; but if he can be content to begin with doubts, he shall end in certainties." **Francis Bacon**

1.1 Preparing for the journey

The prevailing popular image of computer science is of exciting and colourful multimedia, virtual reality and information super highways. The move of computing power from a central inaccessible mainframe to a home consumer appliance has given the computer user the power to view and interact with any of these exciting new forms of media and communication. This is an enticing possibility, and people have responded by purchasing computers for use in their homes (some 34% of homes in the USA now have a home computer [San96]). Anyone interested in investigating beneath the surface, however, is likely to encounter the bewildering world of tools like C++, Perl and link libraries.

Whilst there is an obvious need for tools such as C++, there are potential computer programmers who would now be programming if more intuitive and perhaps more encouraging programming tools were available. This is where direct manipulation [Shn93] and the graphical nature of interface builders are beginning to find a role. Just as spreadsheet applications ignited enthusiasm for programming in the finance world [Nea95], systems such as HyperCard [Com95] and Visual Basic [Mic97b] are allowing other users to see that they might be able to program after all.

These commercial "visual" programming environments enable novice programmers to quickly create simple applications with highly graphical user interfaces. To achieve this, the program-

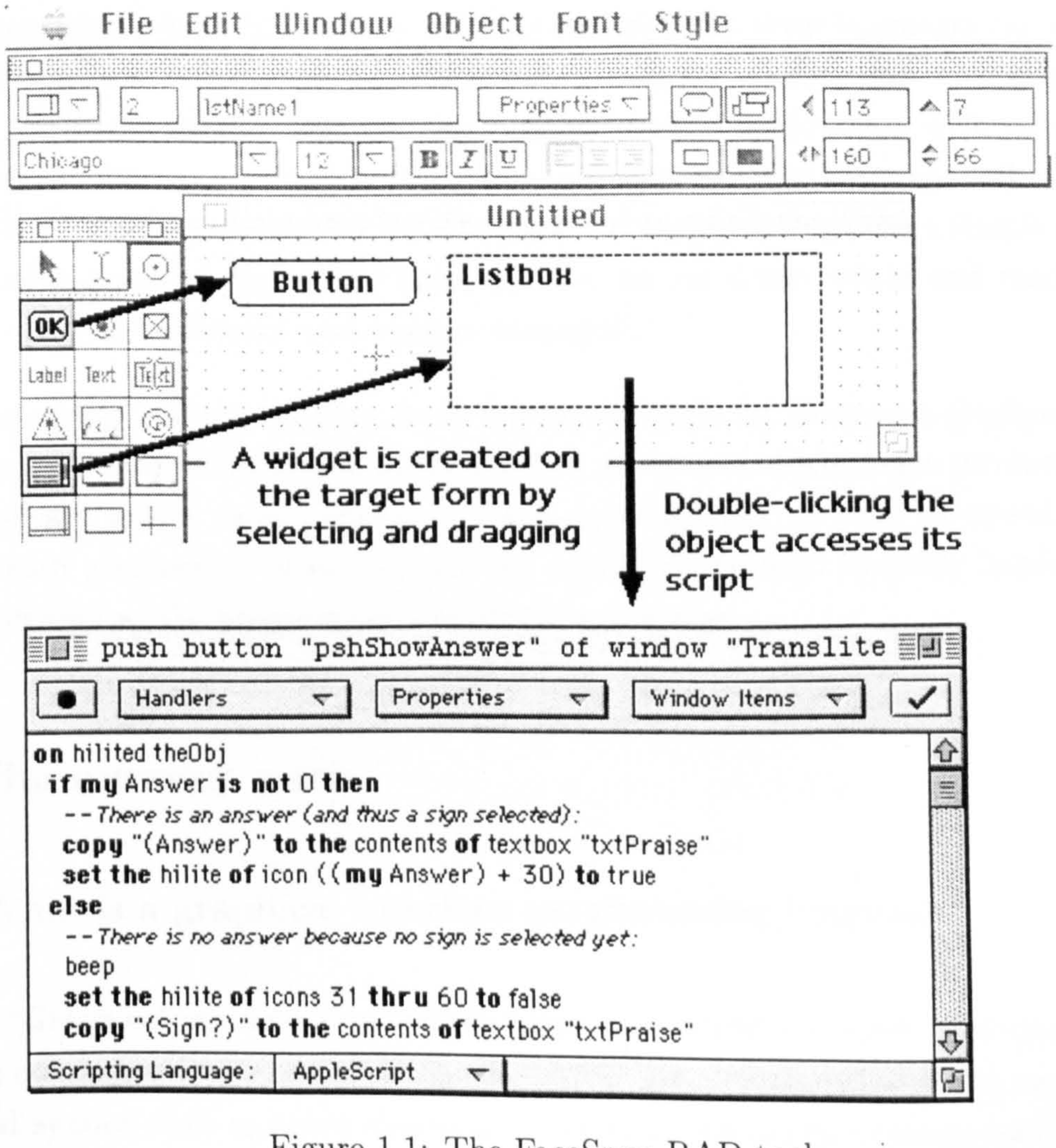


Figure 1.1: The FaceSpan RAD tool

mer is presented with a graphical environment which allows them to assemble an interface using a pointing device with which they select the components they require and place them on the screen. Once the interface is complete, the programmer can define the application's behaviour by writing instructions in a simple, "script" language. See Figure 1.1.

There is, of course, a compromise. In an effort to make the underlying language easier to understand, many of the more complex features, such as dynamic memory allocation and full object orientation are removed. This causes problems as users become more confident with the programming environment and try to create large, or more ambitious applications. Until they encounter restrictions in the language for themselves, there is no way of knowing where those restrictions are. At best, this leads to harmful (as in GOTO considered harmful [Dij68]) compromises in the software design. At worst, it can simply mean that the solution is impossible to implement.

Software engineers and computer science professionals, however, have produced more compre-

hensive languages, such as Ada and C++. If applied properly, these languages can successfully implement large and complex pieces of software. To create applications with graphical user interfaces, however, requires that the language be used in conjunction with an external library or toolkit. The compromise in this instance is that both the language and toolkit are so complex, that a novice programmer has little hope of implementing even a simple application without learning a good deal of the language, how to use a text editor and reading several reference tomes on the toolkit and window manager[1].

The central aim of this thesis, then, is to explore methods for creating a graphical interface programming system which combines the positive attributes of languages produced for both experienced and novice programmers — a system which can be used to create simple applications with a minimum of experience, but which can be used to safely implement more complex software as the programmer's experience increases.

1.2 The current path

1.2.1 What is a graphical interface programming language?

The term "Graphical interface programming language" identifies those languages which can be used to create applications with GUIs (Graphical User Interfaces); not languages based on a graphical syntax, such as those described in [Shu88]. Formerly, systems used to program graphical interfaces were termed UIMS (User Interface Management Systems) [Pfa83], which was confusing as the term referred to anything from simple run time interface management systems, through to complex development environments with their own programming languages. As this latter category of systems became more common, UIMS evolved [HMSS88] into UIDSs (User Interface Design Systems [Mye89]) to emphasise the difference. More recently there has emerged a new flavour of UIDS, the **RAD** (Rapid Application Development) [Mar91] tool, brought about by the increasing popularity of graphical user interfaces on personal computers. Throughout the rest of the text, the acronym RAD will be used to refer to the type of system this thesis is concerned with.

N.B. Just as with UIDS and UIMS, there are no definitive criteria for classifying a tool as RAD: it is, however, the most accurate term available at the time of writing. This does not imply that **the needs of the novice programmer are not necessarily the same as those of someone wishing to prototype an application** — it is simply that the class

[1] More typically, the programmer will require still further skills in, say, integrating a toolkit written in Pascal with an application written in C++!

of tool to be explored, which was designed for the novice programmer, has been hijacked and labelled by application prototypers [OHS89].

1.2.2 Putting the novice in control

Who are the novices?

There was a time, after the introduction of the "Home" computer when every user had to be able to program before their computer could be made to do anything useful. For a time, the terms "user" and "programmer" were synonymous. Subsequently, however, came software packages such as Visi-Calc and WordStar which allowed users to perform useful tasks without the need for them to do any programming. As a consequence of the hardware on which the software ran, these packages were basic, offering limited functions, which the user could easily learn. The computing world started to diverge into programmers (who wrote application packages) and users (who did no more than use them). As the power of the personal computer continues to increase, however, the distinction between programmer and user is starting to blur once more.

Traditional software packages, such as word processors and spreadsheets, in an effort to utilise the enhanced machine resources and keep ahead of their competitors, have grown to become large and complex — often referred to as *creeping featurisim* [UTMP93]. To use a package to its full extent, users will need to learn macro, query or other "script" language. See Figure 1.2 for an example of the Microsoft Word macro language.

The ability of computers to manipulate images, sound and animation has also greatly increased the potential applications of computers to the point that it is impossible for software engineers to cover potential uses with generic, fourth generation language (4GL) type programs. This gap requires that increasing numbers of end users must create their own, custom built, software. This is the class of user which will be considered within the thesis: those with no formal training in how to program, but who find that it is the only way to solve their particular computing problem.

Novice programming - what shall we do today?

"End user programming" is the term most commonly used to describe the programming activities of the end user. A more accurate term for the activities peculiar to this thesis, however, is **novice programming**, i.e. programming by those with no other previous programming

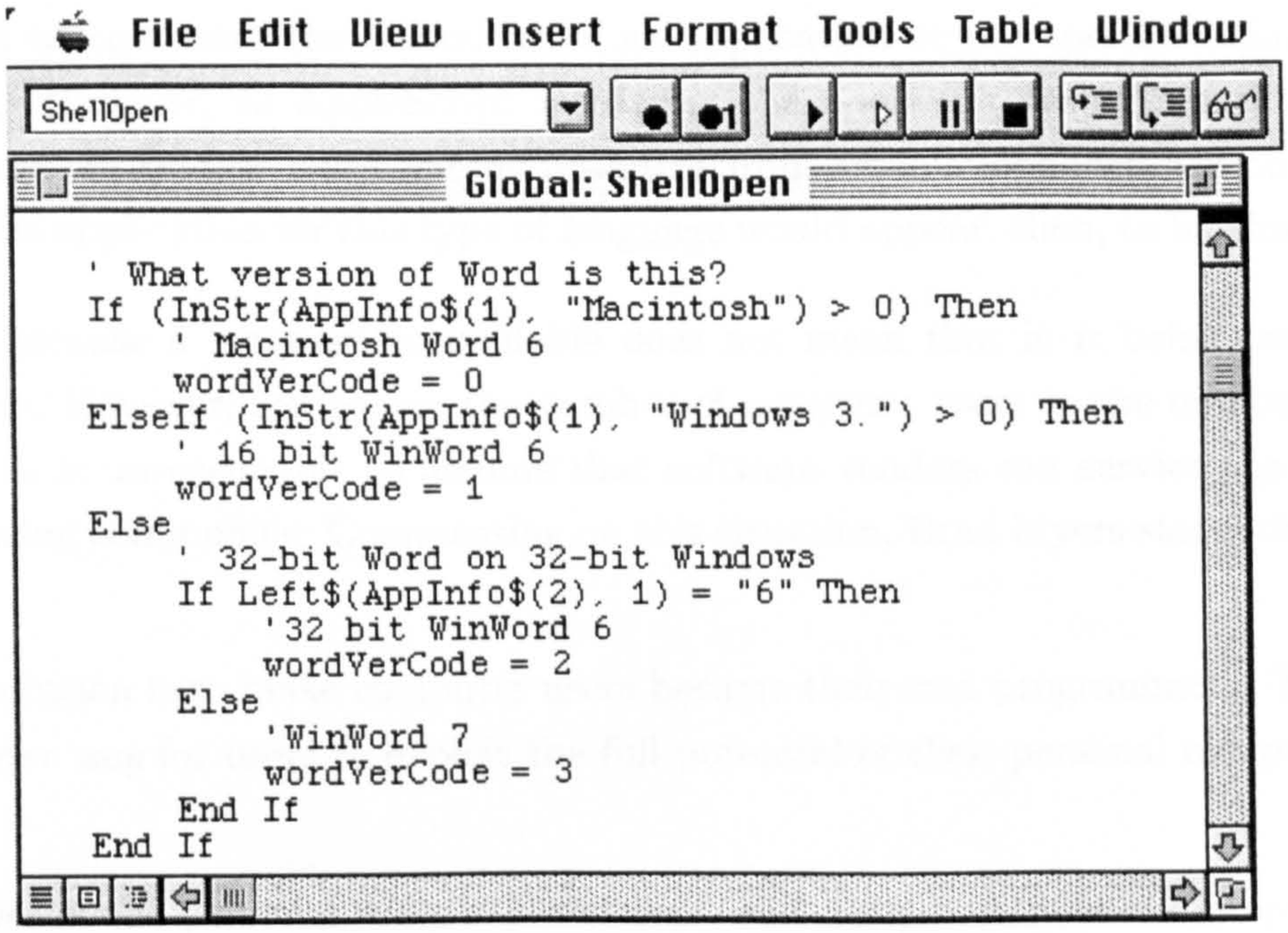


Figure 1.2: Microsoft Word Macro Language

experience[2]. This is a much larger group than might, at first, be expected.

Visual Basic, Microsoft's novice programming language, is the largest selling development language available for Windows machines (included as part of Microsoft Office). Apple has also acknowledged the existence of this group of programmers by the development of HyperCard and HyperTalk, which have been available in some form on every Macintosh since 1991. These systems allow novice programmers to create, should they so desire, completely new applications which have graphical interfaces. Besides creating new applications, novice programmers can also use these languages to customise and use the functionality of other applications [MC97].

Microsoft now include a reduced version of Visual Basic (VBA - Visual Basic for Applications) with Word, Access and Excel (replacing the macro language of Excel 4) and plan to launch Visual Basic Script [Mic97c] as a script language for Internet Explorer. This allows the user not only to automate common tasks, but to develop customised versions of the larger application for specific purposes. Also, included with every version of the Macintosh OS since 1993, is the language AppleScript [Goo93]. This is a flavour of HyperTalk, which is effectively a scripting language for the operating system. Like VBA, an AppleScript program

[2]On a Unix system, end user programming often refers to writing shell scripts — an activity best left to expert programmers!

can be used to customise the behaviour of an application or use specific capabilities of an application. However, as AppleScript is part of the operating system, it can be used to customise any application and combine resources from applications installed on a particular machine. The application for this type of language would appear, then, to be almost limitless.

Of course, because a language is available does not mean that it *is* being used by novice programmers. However, comparing the number of computer users to the number of software developers, it is unreasonable to assume that software vendors can service the needs of the computer-using community. Commenting on this situation, Brad Myers states that [MCH92]:

> The solution is to make computer users become their own programmers. There is *no other way* for users to exploit the full potential of their personal computers.

As the power of the personal computer increases, and users become more interested in technologies such as the Internet and multimedia, the gap between user software requirements and applications available, can only widen. In fact, the World Wide Web is now awash with its own brand of scripting languages (Java [Mic97e], Java-Script [Fla96] and Active-X [Mic97a]) as the software developers realise that they cannot provide all the applications users require, but try instead to sell tools for users to create their own.

1.3 Why take an alternate route?

1.3.1 What novice programmers are being provided with

At the outset, it should be stated that these RAD tools provide substantial benefits for novice programmers and have become known as [Lin95]

> **The microwave ovens of the programming world** — they're new, they're fast and they will probably make a lot of people's lives easier.

Language vendors, however, face a trade off between creating a new language for writing GUI software and trying not to alienate experienced programmers familiar with third generation languages. Hence the current spate of *Visual* languages, consisting of a traditional, text-based language, wrapped in a visual shell. Whilst the language itself may be fairly mature, with no (or possibly a few, well understood) problems, the effects of new paradigms created by enhancing the language have not been so well considered. A simple illustration of this point

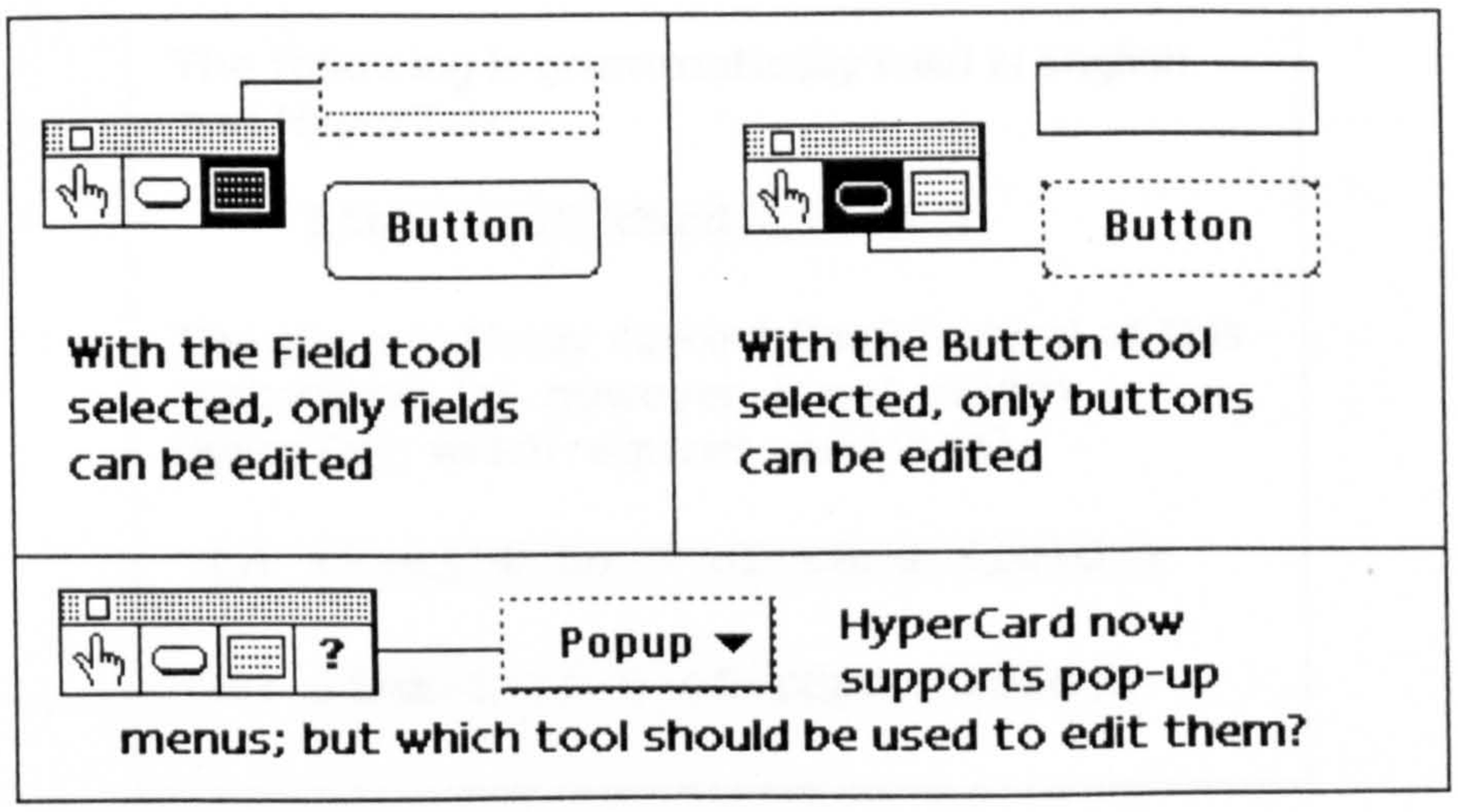


Figure 1.3: HyperCard modification tools

can be found in HyperCard: the 'Button' and 'Field' tools no longer bear a direct relationship to the toolkit they support (it now contains objects other than simple buttons and fields) as in Figure 1.3 — it would be better to have provided a single 'Modify' tool to replace them both. This type of problem could be eradicated if, for example, a more abstract or semantically focused view were taken of the toolkit. Throughout the text, further examples of this type of problem are examined to provide guidance on how to eradicate them from future tools.

The developers of these tools seems to be driven by new technology, without sufficient regard for computing science, software engineering or the cognitive consequences for the end user. In an effort to make a tool "easy to use," designers are employing techniques which can ultimately make the tool *harder* to use. Again, taking HyperCard as an example, its programming language, HyperTalk, has a syntax very similar to English. Whilst this seems familiar and welcome to the novice programmer, it makes for an overly verbose syntax which is beset with the types of ambiguity and confusion found in natural languages. Examples of this type of problem are given in Figure 1.4[3]

In general these RAD and script languages are designed to be less complex than traditional third generation languages, which is a laudable goal. What is worrying, however, is the way in which they are simplified. Ideally, the process of simplification should be driven by a desire to remove unnecessary and duplicated concepts from a language, to provide a more coherent and integrated solution. What is actually happening is the removal of concepts which were traditionally considered complex. As a consequence HyperTalk does not have data structures and Visual Basic has no form of dynamic memory allocation.

[3]A more cynical and extremist view would be that adding features such as a graphical interface to the language is partially a marketing ploy — by using the graphical interface builder, novice programmers could be enticed into buying the product before encountering the underlying language.

The following is grammatically valid in English and HyperTalk:

```
item 1 of card field 1
```

The grammatically correct English plural of this statement, (a), however, is not valid in HyperTalk; which requires version (b)

```
(a) items 1 to 5 of card field 1

(b) item 1 to 5 of card field 1
```

Figure 1.4: HyperTalk and English confusions

By making this restriction to Visual Basic, programmers cannot create applications which, for example, create new buttons and other widgets on the fly — this is something a novice programmer could not have anticipated. By having no data structures in HyperTalk, programmers are forced to implement their own, using tricks such as making a list data structure from a comma separated string. For example in [CT92], the authors describe how to imitate the effects of arrays and pointers using cards, fields and text "chunks." This not only creates overly complex and unreliable programs, but means that the programmer will need to throw away this "knowledge" should they progress onto languages which do not have these limitations. Clearly, this is not the best way to create a language for novice programmers.

In case this starts to sound like a unfounded tirade, it is worth looking at what James Martin wrote about fourth generation languages in 1986 [Mar86]:

> The lack of computer scientists' interest in or knowledge of fourth generation languages is harmful. Most such languages are being built without theoretical foundation.... Most fourth generation languages today are being created and improved by craftsman-programmers with no interest in the theory of languages.

This theme is also continued by Shneiderman [Shn85] who states that

> ...spreadsheets are to the 1980's what COBOL was to the 1960's. The computer science community has largely ignored this fundamental and important innovation

As has already been stated, the RAD language is really a type of fourth generation language and there is no evidence that computer scientists have taken any greater interest in them

than was reported in 1986. An exhaustive review of the HCI literature will show very few direct references evaluating the usability of RAD tools (using libraries, CD-ROM and various Internet search engines, only three such papers were found [Nie91],[TCJ92],[Gre90a]), any article mentioning any of these tools were of a similar type to [HS91]; i.e. reporting the evaluation of some prototype built with the RAD tool. Examining popular press journals (such as Byte), however, will show the huge impact these tools are currently having on the working programmer.

1.3.2 What tools novice programmers should be getting

As was discussed in the previous section, it is inevitable that the output of software vendors cannot meet the requirements of the user. If we are to avoid the worsening of the *software crisis*, then computer scientists and software engineers must start to examine this class of language. By using techniques familiar to both disciplines it is possible to improve both the scalability and robustness of RADs so that novice programmers can implement reliable applications. Quite apart from that, if the usability of these languages is to be improved, it must be founded on a sound, software engineering base — something which cannot be said for the current batch of RAD tools.

Languages which are simplified by the arbitrary removal of complex features are not an acceptable solution and it is not necessary to take this approach. Spreadsheets, for example, can involve complex formulae and functions, yet novice users need not encounter or understand these features to carry out simple tasks. In other words, the spreadsheet was designed so that new users could start to use the application at a very simple level and encountered complexity only as they required it. The next generation of RAD tool must be designed along similar lines, so that the novice programmer can easily create simple applications (something current RADs achieve) but RADs must also contain the capability to produce more complex software as the programmer's needs and experience increase. This is in keeping with Alan Kay's principle of [Kay82]:

> Simple things should be simple; complex things should be possible

Another important aspect to consider is the engendering of good programming practice. Given how ubiquitous programming has become, it would be naive to assume that every programmer will receive instruction in software engineering. Just as the removal of the GOTO statement from third generation languages has undoubtedly led to improvements in software quality [Bro82], it may be that some corresponding alteration to RAD tools will force novice programmers to build inherently better programs. Studies, such as [LH87] which investigate

"first" programming languages indicate that the first language is fundamental to the development of good computing science practice. Dijkstra, in his inimitable way, takes this point to extremes [Dij75]:

> It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.

Although outside the aims of this thesis, it is important to realise that improvements in RADs will impact the expert programmer as well as the novice. For example, as the rate of change in commercial software increases, the software engineering notion of building a prototype, throwing it away and re-implementing the system from scratch is losing popularity. RADs would seem to permit developers to transform the prototype program into a final product, as advocated in the Dynamic Systems Development Method [Con95]. If it is impossible to stop developers from doing this, then computer scientists would seem to have a responsibility to produce RADs which aid the construction of "correct" programs.

1.3.3 Realising ideal tools

In order to examine the domain of tools available to the novice programmer, the next chapter explores the different categories of tool available to the novice programmer. An exploration of two of the most popular RAD tools is then undertaken as a representative sample of this class of tool. This chapter also attempts to define a framework in which these tools may be compared and classified.

The third chapter starts by defining what are desirable attributes for an ideal novice programmer's system. The attributes of the systems examined in chapter two are then examined to see if they are complementary to providing a novice's programming tool.

Chapter four discusses shortcomings in the current systems and proposes designs for new systems which overcome current limitations.

Chapter five then, describes various implementations which attempt to determine the viability of the ideas and solutions previously proposed.

The final chapter will discuss how the work could be furthered and provides suggestions for anyone else wishing to conduct a survey of RAD tools.

1.4 Are we there yet?

The aim of this work is to produce principles and ideas for creating better languages for the novice programmer. Of course, the only way to know if this has been achieved is to set the criteria of "better." Various suggestions have been made on how to recognise a better language ([Mye91] and [Shn85] for example) once it has been created, but less guidance is given on how to arrive there. In this case, three guiding principles shall be set to ensure that the desired path is adhered to. These are:

1. **Parsimony of concepts**: Languages can rapidly become complex, meaning that the programmer spends longer learning the complex rules of the language rather than focusing on the solution of the underlying problem. This is a recognised problem in "text only" languages [Mor93], and is rapidly exacerbated by the nature of RAD tools. *We shall aim to keep the number of concepts within the language to a minimum.*

2. **Programmers are humans too**: There is a temptation to produce tools optimised to a particular class of programmer: novice, professional, student etc. The novice programmer should, ideally, be provided with an environment which expands with a programmer's skills. *We shall aim to remove language features which offer short cuts to the novice, but which will not ultimately become a hindrance.*

3. **Concept re-use**: When creating new classes of language, there is a temptation to believe that the research and underlying principles of previous language generations are no longer relevant. This is not always true, and through re-use, the benefits of the refinement of older concepts can be carried through to new languages. *We shall, investigate the re-use of well understood concepts rather than creating entirely new language features.*

This is not intended as an exhaustive list, but rather gives some insight into the design goals being explored as part of this work. As Meertens puts it in his discussion on language design[Mee81]:

> Composing a language is not merely a matter of putting ingredients together and stirring till the result is a smooth paste.

In the same paper, Meertens also bemoans the lack of any method for language design. Rather than focusing on the ingredients then, we are left to use the above points as a general recipe and to add ingredients as required. To be deemed successful here, the solutions met must exhibit elements of those points listed above.

Chapter 2

Survey of End User Languages

Presented here is a definition of the RAD tool and a survey of the types of systems which might usefully be examined. From these systems, we will extract topics for consideration to which we can successfully apply research from other, closely related, fields.

2.1 What there was

The invention of the microprocessor led to the creation of the hobbyist or "home" computer. In the late seventies and early eighties, there was a flood of eight bit computers from companies such as Apple, Atari, Sinclair, Amstrad etc. aimed at the home market. Almost invariably these machines contained a dialect of BASIC in their ROM — sometimes this was *the only* software available for the machine.

The main factor in the decision to use BASIC was its size. Because BASIC was designed to be small and interpreted [Kur81], it was possible to fit it in the limited ROM capacity of these machines. BASIC therefore became the *de facto* standard for end user programming on the home computer. Besides the hardware considerations, there are other reasons (more relevant to this thesis) for BASIC's success:

- Using BASIC, it was possible to implement any program which the hardware was capable of executing. For example, BASIC did not need to cope with bit mapped displays and mouse events, as it ran on a machine which could only cope with text input and output. BASIC did not need to be *crippled* in any way to remove complexity; it was as complex as the hardware could support.

- BASIC is an encouraging language to use. It is very easy to create a program which responds to user input, or displays output on the screen. The value of this type of responsiveness is vital in improving interface usability [NL95] to promote exploration and confidence; it seems only logical that the same attributes are necessary for a novice exploring and learning a programming language.

- It could be argued that, in a rudimentary way, BASIC provided extensibility for more experienced programmers. Using commands such as PEEK and POKE or assemblers, BASIC programmers who required more performance, or access to specialist hardware, could include assembly routines directly in their programs.

- Another benefit of BASIC was that it was designed as a teaching language and so had perhaps a stronger appeal than the more mathematically orientated languages, such as Forth [Man97] [1] and APL[GA84], which were available at the time.

With the increase in personal computer hardware capabilities, however, BASIC, as it existed on the original home computers, is no longer a sensible choice of programming language.

Yet, currently no end user programming language combines all of BASIC's attributes as listed above. Systems such as Visual Basic and HyperCard provide "instant gratification" [MCH92] and are relatively easy to learn, but cannot implement every class of application and are extensible in very limited ways [MC97].

Finally, it is worth mentioning that BASIC is still being used for end user programming on palm top computers and PDAs (Personal Digital Assistants). Like the original home computers, these machines have limited hardware and can only support quite constrained forms of interaction. Currently available examples include: OPL for the Psion Series 3 which is a variant of BASIC built into the ROM of each machine[Psi93]; and in the case of Apple's Newton [Com97], the built in NewtonScript is being superseded by the much simpler NS-Basic[cor95].

2.2 What is currently available

There is a huge range of tools available for creating graphical user interfaces. The number and type of these tools has grown so quickly that a single, generic classification such as **User Interface Management System** (*UIMS*) [Pfa83] is no longer adequate to describe the diversity of tools available.

---

[1] One machine, the Jupiter Ace, was supplied with Forth. Despite outperforming equivalent BASIC machines by a factor of four, the machine was discontinued within a year, due to lack of sales.

Therefore, we shall create a finer grain classification which will aid in identifying the subdivision of tools this thesis is concerned with. This will serve both to focus the thesis and (hopefully) allow the reader to identify essential attributes of RAD systems and where they fit in the wider spectrum of tools.

The first distinction that can be made is between *active* and *passive* tools. Passive tools are really little more than code libraries, which need to be accessed within some other development environment before they can be used to create interfaces. Roughly speaking, these are available in *developer* (toolkit) and *user* (component) flavours. Because most personal computers are controlled through graphical user interfaces, these toolkits and components are an integral part of most operating systems. Active tools are built on top of the widget set and allow the programmer to specify the widgets' run-time behaviour. The active tools can then be classified in terms of the degree of control over passive resources each tool affords to the user.

2.2.1 Passive tools

Toolkits

Toolkits, or APIs (Application Programmer Interfaces) are programming language libraries containing common code routines used in creating graphical interfaces. These routines typically implement interaction devices (widgets) allowing values to be entered or displayed on the interface; e.g. buttons, fields and slider bars. By using toolkits, application programmers can save on the amount of code that needs to be rewritten and provide consistent interface elements within different applications. The use of toolkits in programming applications has resulted in an estimated ten fold increase in programmer productivity [Mye94].

Unfortunately, these benefits cannot be felt directly by novice programmers. Toolkits exist as libraries for programming languages, and must be accessed through these languages. There is some flexibility in that other languages may access the toolkit routines, provided they support the data types of the routines' parameters[2]. Other possible problems with toolkits include:

- difficulty in creating the toolkit initially (supplier problem)
- any errors in the toolkit will be propagated to all client applications (supplier problem)
- difficulty of recalling numerous procedure names and their parameters (user problem)

[2]In order to access Dynamic Link Libraries (DLLs) Visual Basic supports C data types such as "floats" and "doubles".

- difficulty in customising widgets (user problem)

For example: To add a new widget from a toolkit, the programmer would first include the library containing the routines into their program, then invoke a procedure (or send a message if the toolkit is object oriented) similar to the one beneath:
`NewControl(theWindow; boundsRect; title; visible;value; min; max; procID; refCon)`
Other procedures (e.g. `MoveControl()`) can be invoked to alter the widget, whilst still more are provided to read and alter state variables associated with the widget (e.g `SetControlValue()` and `GetControlValue()`). These examples are all taken from the Macintosh System 7 toolkit [Com92].

Components

As languages have developed from third to fourth generation, and object orientation gains popularity, so toolkit widgets have developed into components. Although still interaction devices, components are the encapsulation of the properties and methods of that device into an individual object to provide a higher level unit of interaction [3].

Components are usually designed to adhere to some form of interface standard, permitting their use as a resource in a run time environment. This run time environment is usually an interface builder, allowing the object to be used at a higher level than a widget from a toolkit. Programmers are thus able to specify run time behaviour of components by interactively setting design time parameters as in Figure 2.1. In contrast, the behaviour of toolkit widgets can only be defined through creating and assigning event handlers from within program source code.

Examples of components include Window's VBXs (Visual Basic eXtensions) [4] and XCMDs (External Commands) on the Macintosh. XCMDs were designed to be used by HyperCard and were also adopted by its clones (such as SuperCard). VBXs were primarily created for Visual Basic and any application using VBA (Visual Basic Application: a cut down version of VB integrated into some PC applications). Other development languages such as Visual C++ [Mic97d] and Borland C++ [Cor97a] support VBXs, although, as compiled languages, they do not have a run time environment to support components as fully as Visual Basic.

The specification of component interfaces opens the way for the benefits of object orientation

[3] Some toolkit libraries have had object wrappers placed around them; for example, the Think Class Library [Bea93] built on top of the Macintosh toolkit.

[4] With the advent of Windows 95 and Windows NT, the VBX is being replaced with the 32-bit OCX.

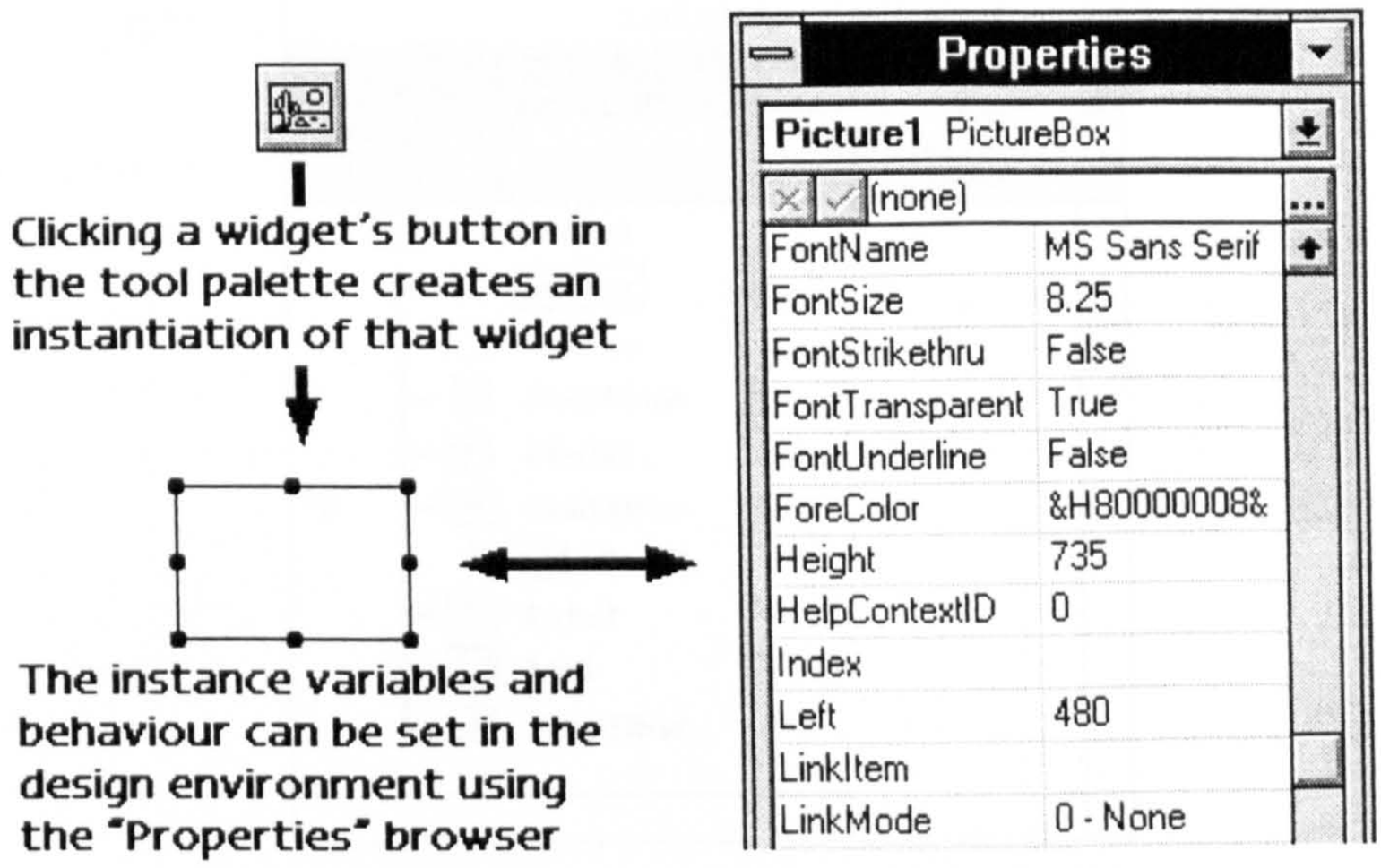


Figure 2.1: Using components

such as:

- Errors in a component from a given set can be overcome by replacing the offending component. Toolkits typically require a reissue of the entire library.
- New and specialised components can be added to existing software, allowing it to be used in a wider variety of applications, e.g. adding an interaction device, allowing physically impaired people access to software.

At present, only a few applications utilise components, but future standards such as OLE 2.0 and OpenDoc should allow the benefits of component software and applets to be fully exploited[5].

Components allow the benefits of toolkits to be accessed in a more convenient form, making more efficient use of the experienced programmer's time and allowing the power of toolkits to be accessed by those less experienced. No longer is it the responsibility of the programmer to remember the events and attributes of each component they need to use.

Interoperability

Because components and (especially) toolkits are such low level tools, there is little commonality across platforms. One solution to this problem is translation tools or cross compilers

[5] A comparison and discussion of OLE and OpenDoc can be found in [IBM97].

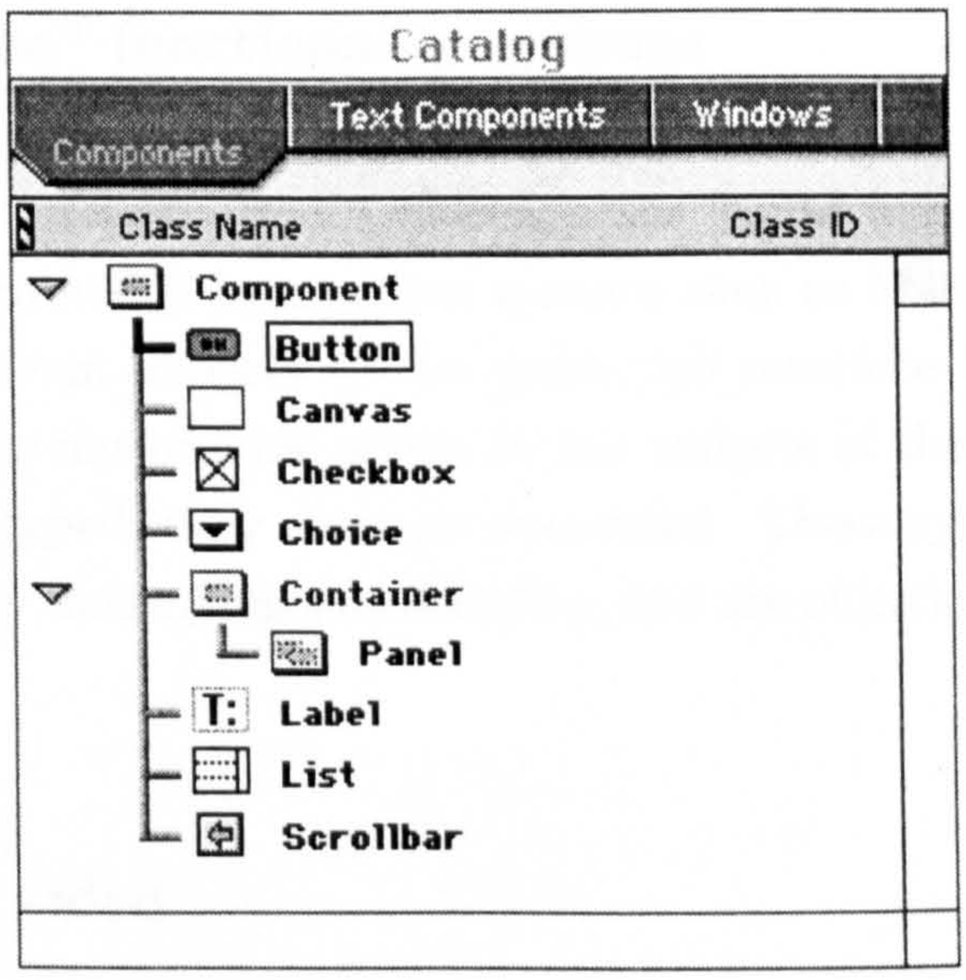


Figure 2.2: Only the lowest common denominator widget set is supported.

which translate calls from one toolkit to another. An alternative approach, is to define a simple toolkit which can execute on all platforms. Whether this is done explicitly to provide cross platform compatibility (as with Java [Mic97e] and tcl/tk [Ous96]) or to provide a simple system for teaching (such as SUIT [PYD91]), these toolkits provide only limited functionality. As the toolkit is inevitably built to the lowest common denominator of hardware from among the various platforms supported, the interfaces implemented are often crude and unsatisfactory. See Figure 2.2.

2.3 Active tools

Essentially fourth generation languages, there are many types of tool optimised for the creation of graphical user interfaces. Programming a visual system in a completely textual language is both irritating and error prone (without being able to see the interface, the programmer must visualise the interface, its layout and the visual attributes of each widget). Active systems allow programmers to at least design the visual part of their system in a graphical, WYSIWYG environment. At their most basic level, they provide mechanisms with which the user can "draw" an interface. More sophisticated systems provide facilities to define interface functionality and, in some cases, facilitate the creation of the underlying application.

2.3.1 Basic or "zero" functionality systems

Basic systems are used purely to draw and design the "look" of an interface. An appearance of functionality can be provided by complex systems such as MacroMind Director [Mac97b], which can animate relevant aspects of the prototype interface in response to user input. Usually these systems do not provide access to the widgets of the host operating systems — if these are to be prototyped, they must be re-created. These systems, however, provide no way of implementing any underlying functionality, and are of little interest to the work of this thesis.

2.3.2 Graphically aided

There is a class of CASE (Computer Aided Software Engineering) tool which uses graphical browsers to aid code authoring. These browsers do not directly create code and are not available at run time — in most cases they simply provide an alternative view of the code showing, for example, object hierarchy diagrams. These tools are not specifically created to implement interfaces nor provide special features to do so; they are included merely to aid classification.

2.3.3 Interface builders

Interface builders allow access to the operating system's widgets through a graphical environment. A non-programming user can create an interface by selecting and positioning widgets using a direct manipulation interface builder. The output from the interface builder is either:

- A source file in a programming language (usually C) which contains the calls to the operating system's toolkit. The application is then created by the programmer inserting routines inside the automatically generated code. Of course, once some custom code has been added, the interface builder can no longer be used to alter the interface — hardly ideal support for iterative design [Con95]. An example of this type of system is X-Designer for X-Windows [Tec97].

- A file containing descriptions of widgets and resources (such as graphic images and sounds) used in the application. Items contained in this file are allocated a unique identifier and referenced from the source code of the application they pertain to. There is an overhead in remembering the code for each widget, but this approach supports

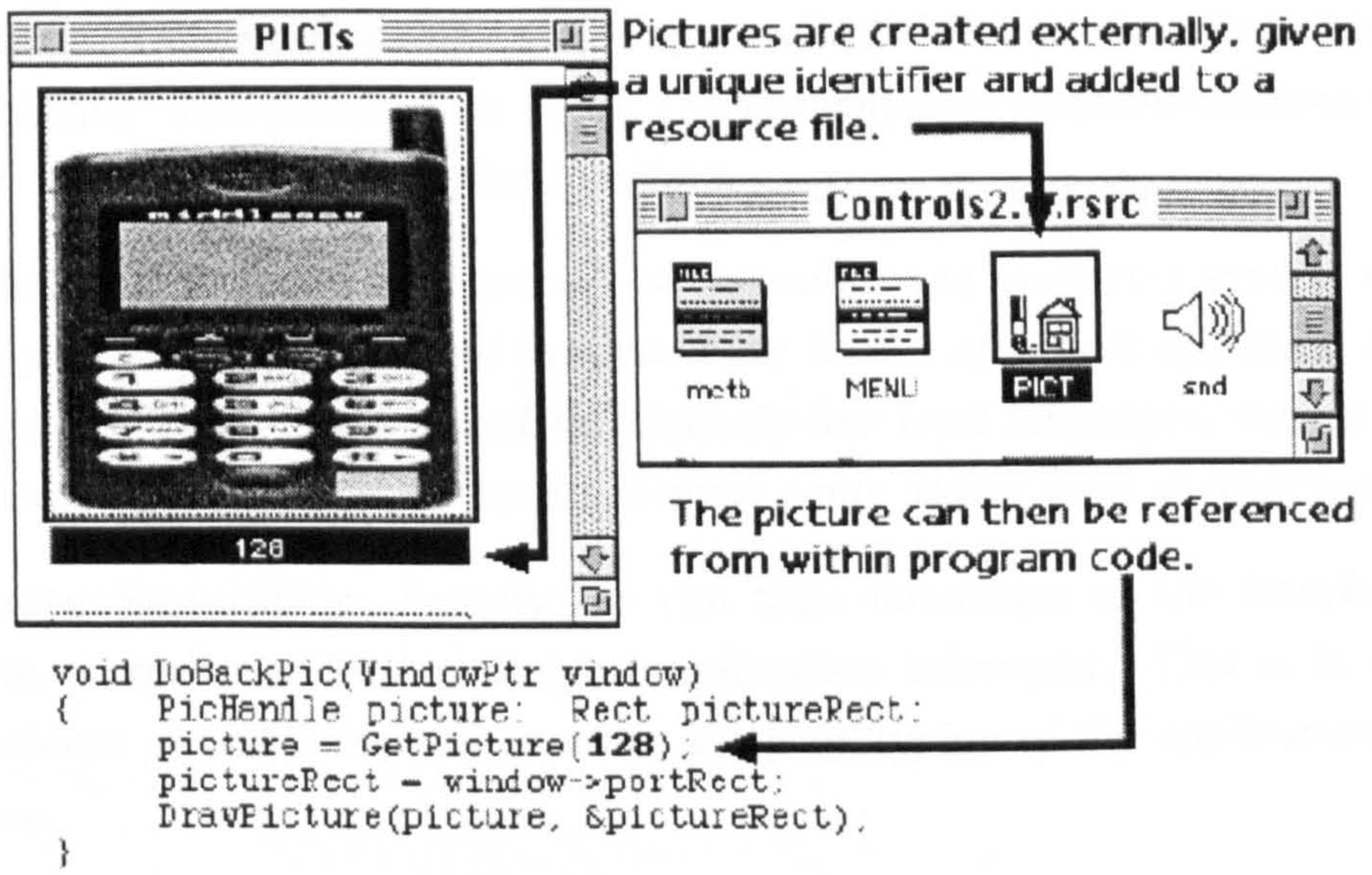


Figure 2.3: Using Res-Edit to build interfaces

the use of the interface builder at any time in the development process. An example of this type of system is Res-Edit from Apple [AS95].

These systems do not support novice programming, but require an experienced programmer to implement the application's functionality. The reason for this is largely historical, as interface builders now tend to be coupled to commercial development languages as a simple way of improving a language's ability to create GUI applications. This allows experienced programmers to retain their experience without needing to re-train for a GUI-centric language.

These tools are either available individually (languages such as C++ and Pascal on the Macintosh use Res-Edit as an interface builder — Figure 2.3), or as part of a larger development environment (Visual C++ for Windows comes with its own graphical interface builder).

2.3.4 RAD — Rapid Application Development

Interface builders were created as a way of updating existing programming language technology. This class of RAD tools, however, are *specifically* created to implement applications with graphical user interfaces. These systems provide ways of specifying the behaviour of interface objects but are not always powerful enough to produce a commercial application.

In order to identify RAD tools, as opposed to languages enhanced with interface builders, we shall use the following criteria which must hold true if a tool is to be classified as RAD. These are:

1. **Integrated:** Consist of one integrated application with, at most, creation and execution environments. *This criterion is set to exclude language implementations which require external text editors and interface builders.*

2. **Complete GUI:** Have no direct access to underlying operating system toolkit[6]. *It is not the purpose of a RAD tool to extend the toolkit of a host operating system. Also, because toolkits are implemented in relatively low level languages, novice programmers should only be exposed to the toolkit through some higher level abstraction.*

3. **Complete Semantics:** Specify the run time behaviour of the interface and be of sufficient power to at least prototype application behaviour. *This is to exclude interface builders — RAD tools must allow the prototyping of the application, not just the interface.*

4. **WYSIWYG Interface Builder:** Support the building of an application's interface through direct manipulation. *There are systems which fulfil the previous criteria, but incorporate a separate interface specification language. This cannot be said to be a true novice programmer's tool as it requires the learning of two languages. Furthermore, one of the reasons the term "RAD" is used to describe these tools is their support for* **Rapid** *prototyping, support not provided through the use of an interface specification language.*

5. **Instant Semantics:** Be effectively based on an interpreted language. *This is not an absolute rule, but is a fairly good guide — RAD tools are mostly used for prototyping short code-compile cycles [Pea88]; execution speed is a secondary consideration.*

2.3.5 GUI development tools for the experienced programmer

Very similar to RAD tools, expert tools can be distinguished by having any of the following facilities:

- Based on a compiled language.
- Direct access to toolkit.
- Facilities to create new widgets.

The distinction between RAD and this class of tool is blurring as traditional software development models such as the waterfall [Som96] and SSADM [GS95] are having to be re-thought

[6]Toolkit widgets are accessible, but as high level objects in the creation environment — programmers should not be able to create new toolkit widgets from inside the environment.

in the light of developing GUI software [7]. RAD tools better support new software development methodologies, such as DSDM [Con95], which promotes iterative development and prototyping. Hence new professional development tools are being built around the RAD paradigm.

The most popular example of this class of tool is Microsoft's Visual C / C++, consisting of an interface builder (as described above) combined with a third generation language. Whilst no doubt improving the lot of the professional programmer, studying this particular class of tool adds little to this thesis as they are inevitably grounded in third generation language technology.

One type of professional development tool which are interesting, however, are those designed *since* the advent of GUI programming. Included in this section are systems, such as Garnet [MGD+90], which was built as a research tool capable of creating commercial quality applications. It cannot really be considered as a true RAD [8] as its underlying programming language is CLOS — despite what emacs users may say about end user programming in lisp, CLOS cannot be considered a suitable language for the untrained novice programmer.

What makes these tools interesting, however, are new approaches to programming, such as *programming by example* [Mye86] and *constraint programming* [Lel88], which are of use in novice programming systems. As this thesis is not directly concerned with this class of tool, a full investigation is not provided. Instead, ideas and paradigms from this class of tool will be discussed in future sections.

2.4 Flavours of RAD

Having now specified how to identify a RAD tool, we shall investigate the various categories which currently exist.

2.4.1 Visual programming languages

Before products such as *Visual* Basic and *Visual* C existed, a "Visual" language was one which had a graphical, rather than a text based, syntax [Shu88][9].

---

[7] Of which, half the code is accounted for in implementing the interface [MR92].

[8] In fact, Myers himself classifies Garnet as a development environment for experienced programmers.

[9] Judging by discussions on the "comp.sys.language.visual" newsgroup, this hijacking of terminology has not been welcomed by the visual language community.

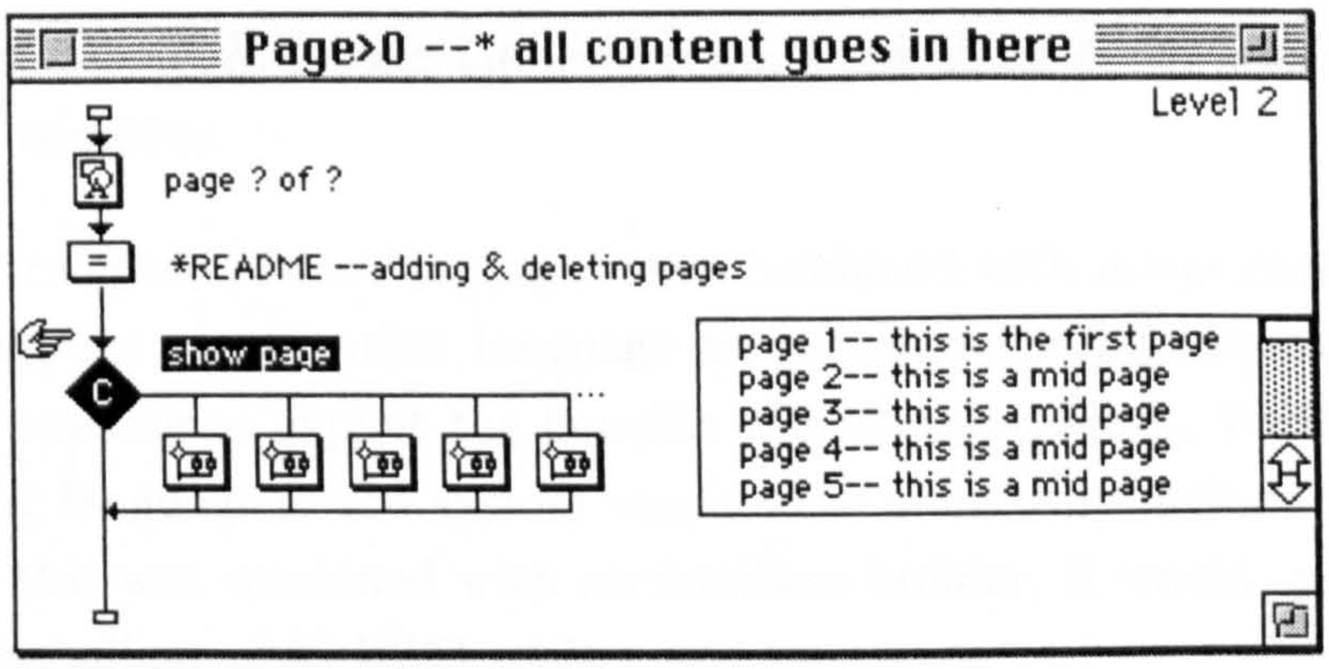


Figure 2.4: The icon "C" above indicates a Conditional loop where the value of some variable moves control flow to one of the icons to the right of the "C." It is interesting to note how the creators of this program acknowledge the similarity to flow charts by choosing the same symbol to denote a decision.

The majority of these systems, however, do little other than convert the syntax constructs of text based languages into graphical equivalents. The result is a language which differs only at the *lexical* level [Bux83]. One of the most popular, commercially available, graphical languages is AuthorWare from MacroMedia [Mac97a]. As can be seen from Figure 2.4, the semantics of the constructs are, in principle, identical but expressed in a graphical notation.

"Visual" languages

The *other* type of visual language mentioned above is exemplified by Visual Basic. These systems consist of a third generation, text based, language which has been amalgamated to a direct manipulation interface builder. In his 1991 survey of user interface tools [Mye91], Myers classifies HyperCard and similar systems as "Frame" tools as they are limited to displaying frames (or cards) which contain information and widgets. This is by far the most common type of RAD tool, with new variations being released, for the present at least, on a continuous basis.

Interface specification systems

One final class of RAD tool worth considering, is that which combine a third generation language with a further, interface specification language. Before objecting that this class of tool is excluded as the interface is not built using direct manipulation, it is worth remembering that these systems were designed when interface builders were not common. These systems are worth examining as they concentrated on modelling the interface at a higher, more problem orientated level than the programming language. This freed the programmer

from making certain widget choices and layout decisions which could possibly be better made by the interface compiler.

In fact, if interface specification languages were combined with direct manipulation builders, perhaps by using the specification language as an internal representation for the builder, then it may be possible to exploit the benefits of both approaches. Further, if the underlying programming language of the system was able to seamlessly include interface specifying primitives, and this was combined with an interface builder, it would, perhaps, present the most powerful paradigm of all RAD tools.

To date, there is no RAD tool based on this paradigm, although the concepts appear individually in other tools. Systems such as Jade [MVZ92] combine a language to specify the interface with a visual editor to fine tune its output. Another tool, IDL (Interface Definition Language) [FKKM89] is a Pascal derivative which is used to specify an application's functionality, but which can then be used to automatically generate an interface to that application.

2.5 Comparison of RADs

It is impossible to provide an exhaustive study of the features of every available RAD tool. At present there are hundreds of these systems available and, seemingly, more appearing every day. However, if these languages are to be improved, then some investigation must be made into the attributes and features common to them all. To that end, an investigation was undertaken into two RAD tools which were felt to be representative of the wider field — namely Visual Basic and HyperCard.

2.5.1 Brief outline of HyperCard

HyperCard evolved from the "MacPaint" application shipped with the original Macintosh. Bill Atkinson, MacPaint's author, gradually added functionality to MacPaint, allowing pictures to be linked with buttons. This development process continued with the inclusion of a general purpose programming language (HyperTalk) in HyperCard version 2.0 and various multimedia options such as colour and animation in the current (2.3.5) version. Consequently HyperCard has a few inherited eccentricities and is too closely linked to the Macintosh environment to ever be ported to another platform.

2.5.2 Brief outline of Visual Basic

Visual Basic 1.0 was released in 1991, followed by 2.0 in 1992, 3.0 in 1993, 4.0 in 1995 and 5.0 in 1997. Although impossible to determine accurately, Microsoft estimates that, judging from their sales figures, there are some 2 million Visual Basic programmers world wide. Visual Basic is only available for Windows or DOS, although VBA is available on the Macintosh.

2.5.3 Why Visual Basic and HyperCard

The aim of this work is to improve tools available to the novice programmer. The two hardware platforms which end users are likely to encounter are Windows and Mac OS based systems. Without doubt, the most popular RAD tool available for the Windows platform is Visual Basic and correspondingly, the most popular tool for the Macintosh is HyperCard. The success of these tools has also resulted in imitators: SuperCard [All97] and News [WR92] were inspired by HyperCard, whilst Delphi [Cor97b] was obviously inspired by Visual Basic. By looking at HyperCard and Visual Basic, it is reasonable to assume that the attributes and deficiencies in the tools supplied to the greatest number of novice programmers shall be discovered.

Of course, if these systems were identical, merely different ports of the same system onto different hardware, there would be little value in their comparison. However, as shall become apparent, they are very different tools, embodying different language design philosophies — Visual Basic was created by joining an existing language to an interface builder, whilst HyperCard is a completely new combination, integrating language and interface builder into one environment.

Comparing implementations of these two differing paradigms will serve to highlight the majority of commonalties and differences in the current batch of RAD tools. The only class of RAD tool excluded by this comparison are languages with a visual syntax. Indeed, as has already been discussed, the only significant difference in these systems is their use of a language which has a graphical syntax. Although the impact of a graphical syntax will be examined in further chapters, it is of little consequence when examining every aspect of differing RAD tools.

2.5.4 How do we compare

One of the challenges in comparing members of a new class of tool is that there will, by definition, be no standard mechanism to compare them. In fact, there has been little previous work in comparing any form of language which incorporates its own environment. As Green puts it in [Gre90b]

> To date, surprisingly little comparative work has been reported on environments, and virtually none that treats the problem of matching environment to notation.

These particular tools combine elements of visual languages, third and fourth generation languages, text editors etc. In order to compare RAD tools, it would seem logical to start by examining work comparing the tools from each of these individual domains.

Third Generation Languages

Not surprisingly, the greatest wealth of literature is found on the comparison of third generation languages (in fact, it seems there are almost as many books and articles on comparative languages as there are languages themselves!) As Wirth notes [Wir84], discovering what should be compared, however, is less straightforward:

> It is manditory to distinguish was is manditory and what is ephemeral...in a computer language

This is not helped by the diversity of types of texts, ranging from those interested in programming semantics [Ten81] through to student texts [WC88]. One text [Bar86], however, provides a comprehensive summary list:

- Program structure: e.g. is the language procedural, object-oriented or of some other structure.
- Data Formatting: data types, data naming, data structures.
- Input / Output: (not particularly relevant as this comparison is constrained to GUI programming systems.
- Miscellaneous Syntactic Issues: Syntax structure, reserved words etc.

Each of these criteria will be used in comparing the languages.

Fourth Generation Languages

Most comparisons of fourth generation languages are of a very applied nature, written for prospective users rather than implementers or computer scientists. Again, this is hardly surprising given that fourth generation languages were created for non programmers. Also, due to the vertical nature of fourth generation languages, surveys tend to identify groups of applications (e.g. report generators, spreadsheets etc.) and compare particular languages within their own application classification. Consequently, there is little that can be gained from this literature as, in effect, the examination concerns an entirely new class of fourth generation language. From using and reading literature on fourth generation languages, common attributes to this class of language include: **code translation, error recovery, level of user customisation, support for different hardware platforms** and **support for new or third party additions.**

Visual Languages

A key text on visual languages [Shu88], like the literature on fourth generation languages, provides a classification scheme for identifying different classes of visual language, but does not provide for comparing systems from the same class. Classes are measured on three scales, namely:

- **Visual extent:** The number of visual representations within the language.
- **Language level:** How pedantically the programmer must specify steps in an algorithm — declarative languages being at a high level and imperative languages, such as C, being at a low level.
- **Scope:** The range of problems to which a language might be applied.

This, and other comparisons (e.g. [LK90]) concentrating on defining classes is symptomatic of the infancy of this particular branch of programming language design.

What is useful from investigating visual languages is the coupling of an editor or design system to the execution system. This results in a single system which exhibits different (design and execute) **interaction modes.** Also, because the design mode / editor mode is integrated, it is possible to investigate **editor support for the language.**

Integration

Finally, there needs to be some measure of how well the various parts of a RAD tool integrate. Essentially, a RAD tool is comprised of a **creation environment, programming language** and a **toolkit**. Each of these interact as follows:

- *Environment - Toolkit*: Using object browsers within the environment, it is possible to instantiate, destroy and modify toolkit widgets.
- *Environment - Language*: The environment allows code to be created and can provide other language support facilities such as debuggers, profilers etc.
- *Language - Environment*: Statements may exist in the language to access or alter the settings of the creation environment.
- *Language - Toolkit*: The language is used to define the behaviour and alter the state of toolkit widgets.

N.B. The toolkit, of course, is a code library and cannot affect the creation environment or the language. Consequently, there are no "Toolkit-Environment" or "Toolkit-Language" headings.

2.6 Comparing Visual Basic and HyperCard

It is proposed to investigate Visual Basic and HyperCard under two broad headings, namely **Language** and **Environment**. These two main heading are subdivided into various individual features: each feature being explained briefly and followed by a description of how that feature is manifest in each of the two systems. It is not intended to give any analysis within this chapter as to what constitutes a *good* or a *bad* feature — such deliberation is left to the next chapter.

2.6.1 Environment

Modes of interaction

RADs have two basic modes of operation: *design* mode and *execution* mode. In the design mode, users may manipulate widgets and specify program and interface behaviour. In exe-

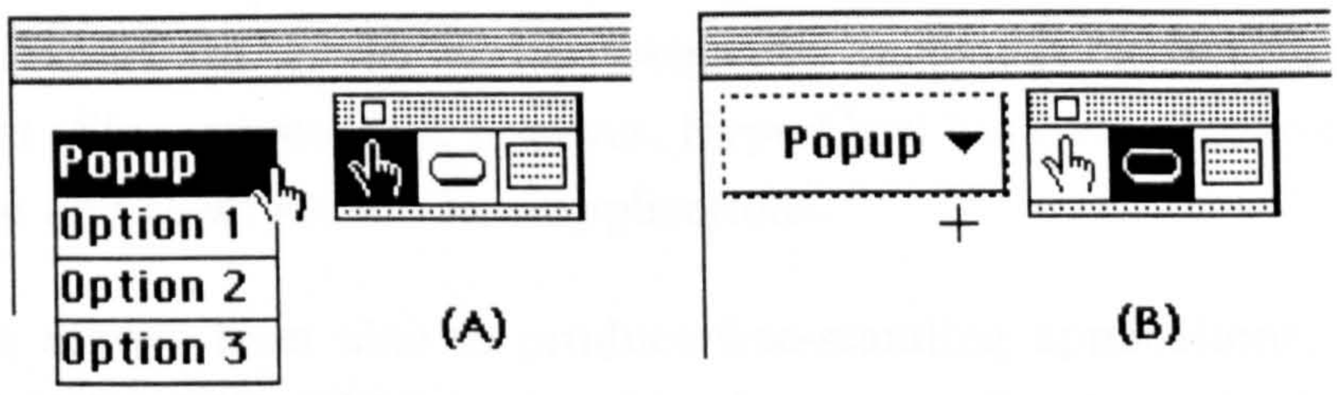


Figure 2.5: In situation "A," with the "Execute" tool selected, clicking on the menu shows the selections. In situation "B," with the "Button Edit" tool selected, clicking on the button selects it for editing.

cution mode, the program is run by its creator (or another user) and interaction may take place.

HyperCard has only one environment, and differentiates between modes by using a different tool, denoted by a different pointer. For example, clicking on a button with the "Button" pointer shows the attributes of that button. Clicking on the same button with the "Browse" pointer causes the click event handler to be executed. In this way, HyperCard promotes the idea that there is only one environment, but that the user can posses many tools (Figure 2.5).

Visual Basic has separate design and execution environments. All coding and widget creation is carried out in the design environment and run time support is provided by the execution environment. Once executing, the design tools cannot be used to alter any aspect of the interface, (although the code editor appears if an execution error occurs). This is a very different model to that presented by HyperCard.

Code translation

Rather than being purely interpreted or purely compiled, HyperTalk and Visual Basic code both use a hybrid "compile on demand" technology. Rather than compiling the whole program prior to execution, each subprogram or event handler is compiled just prior to invocation. This form of compilation is possible because the executing application has the support of the creating environment at run time. Both systems, however, can compile code into a stand alone form[10].

Until version 2.2, HyperCard had no way of producing software which would execute outside its run time environment. Before this, anyone distributing software had to rely on the recipient having access to a full version of HyperCard, or the HyperCard player (an "execute mode only" version of HyperCard). Of course, there still remained the problem of creating a stack

[10] Although in [TCJ92] Thimbleby observes that "few are the HyperCard stacks you would let someone else use."

in a version of HyperCard, newer and consequently incompatible with the version of the intended recipient. Since version 2.2, however, HyperCard has been able to compile programs into (rather large and slow) stand alone applications.

Visual Basic has always been able to produce free-standing applications, on the condition that a file called "VB**RUN.DLL" is placed in the correct directory. Again, the appropriate version of this file must be in place before the Visual Basic application will run. The professional edition of Visual Basic can create installation scripts which place this file and any necessary custom widgets in the appropriate directories.

Extendible widget set

Although HyperCard and Visual Basic are supplied with a set of interface widgets capable of implementing most interfaces, these systems must also provide mechanisms allowing new widgets to be included. New widgets may be required for vertical applications, or be created by continuing research into user interaction. If RAD tools are to protect novice programmers from the underlying toolkit of the operating system, then they must also provide some way to allow expansion by more experienced users.

HyperCard can be extended by the rather awkward mechanism of XCMDs and XFCNs. These are C / Pascal modules which are compiled and included as primitives within HyperTalk. The programmer has no way to access new widgets from within the "design" environment (e.g. it is not possible to add, say, a pop-up menu tool to the palette [11]). See Figure 2.6.

Visual Basic can use VBX components and DLLs. VBXs are usually visual controls accessible from the design environment (Figure 2.6), whilst DLLs are function libraries callable from within Visual Basic code. This provides a high degree of flexibility and has proved to be a key contributor to Visual Basic's success, with many other Windows applications supporting the VBX standard.

Platform support

One reason for using a high level language is to allow code written on one type of machine to be recompiled and executed on another. Whilst this is possible with third generation languages, RAD tools rely heavily on the underlying toolkit, which inevitably varies between

[11] As new versions of HyperCard add new widgets, they must be classed as either *button* or *field* — consequently, in the latest versions of HyperCard, pop-up menus are available from the *button* tool.

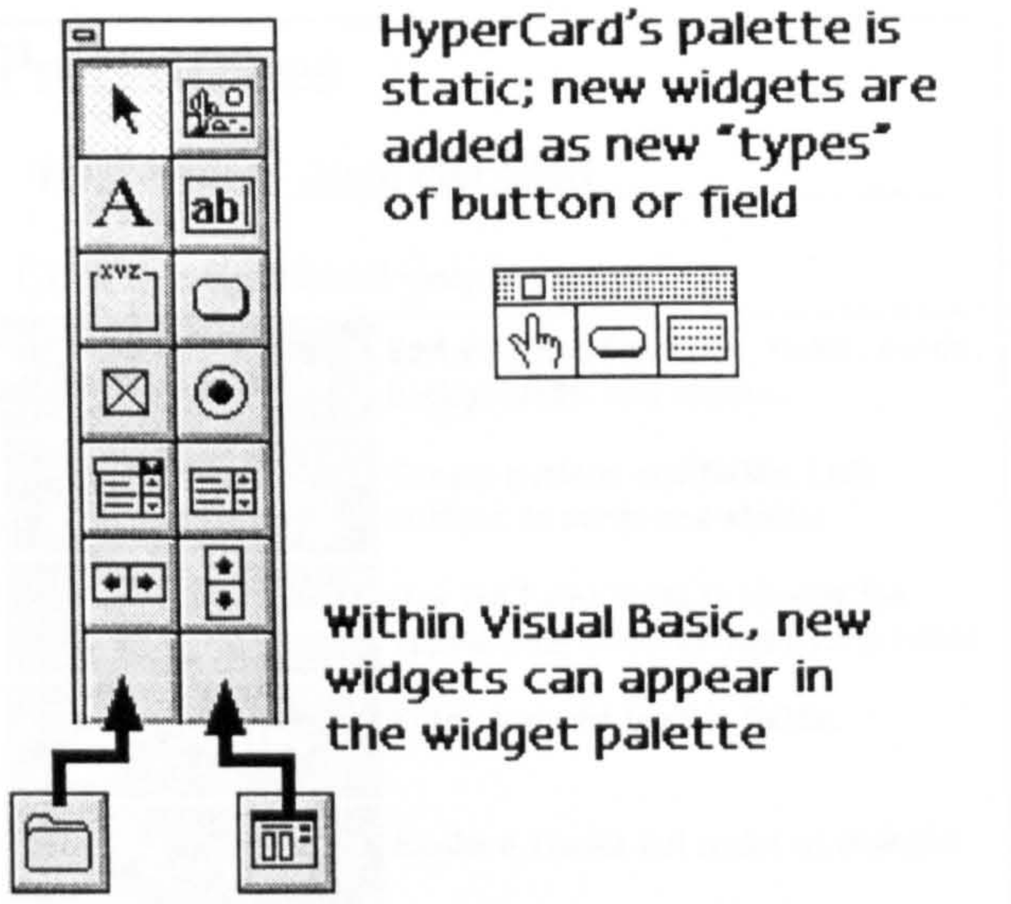


Figure 2.6: HyperCard and Visual Basic widget palettes

different machines. As a consequence, neither Visual Basic nor HyperCard can execute on, or produce code for, more than one hardware platform.

Program Editors

As editors come integrated with RAD tools, they can be optimised for the system's programming language and remove some of the cognitive load from novice programmers. By using automatic statement, bracket or quotation completion, many syntax errors can be removed [Gil86]. Other semantic clues, provided by "pretty-printing" and automatic indentation, can also lessen the chance of creating errors [SSSW85].

The Visual Basic editor can perform an automatic syntax check on any new code and immediately notifies the user of errors. Code understanding is also enhanced by the editor which colours reserved words differently to user variables. The editor also provides automatic sub program termination statements whenever a programmer creates a new sub program.

The HyperCard script editor performs no syntax checking. In fact, the only language aware feature it exhibits is automatic indentation of nested structures such as loops and condition blocks [12]. A possible argument for this lack of support is HyperCard's ability to use different language syntaxes. For instance, it is possible to replace the standard HyperTalk interpreter with an Apple Script interpreter (as included in version 2.2) or a French variant of standard HyperTalk.

[12] The automatic indentation feature is essential for understanding how the interpreter will parse HyperTalk's `if..then..else` statement.

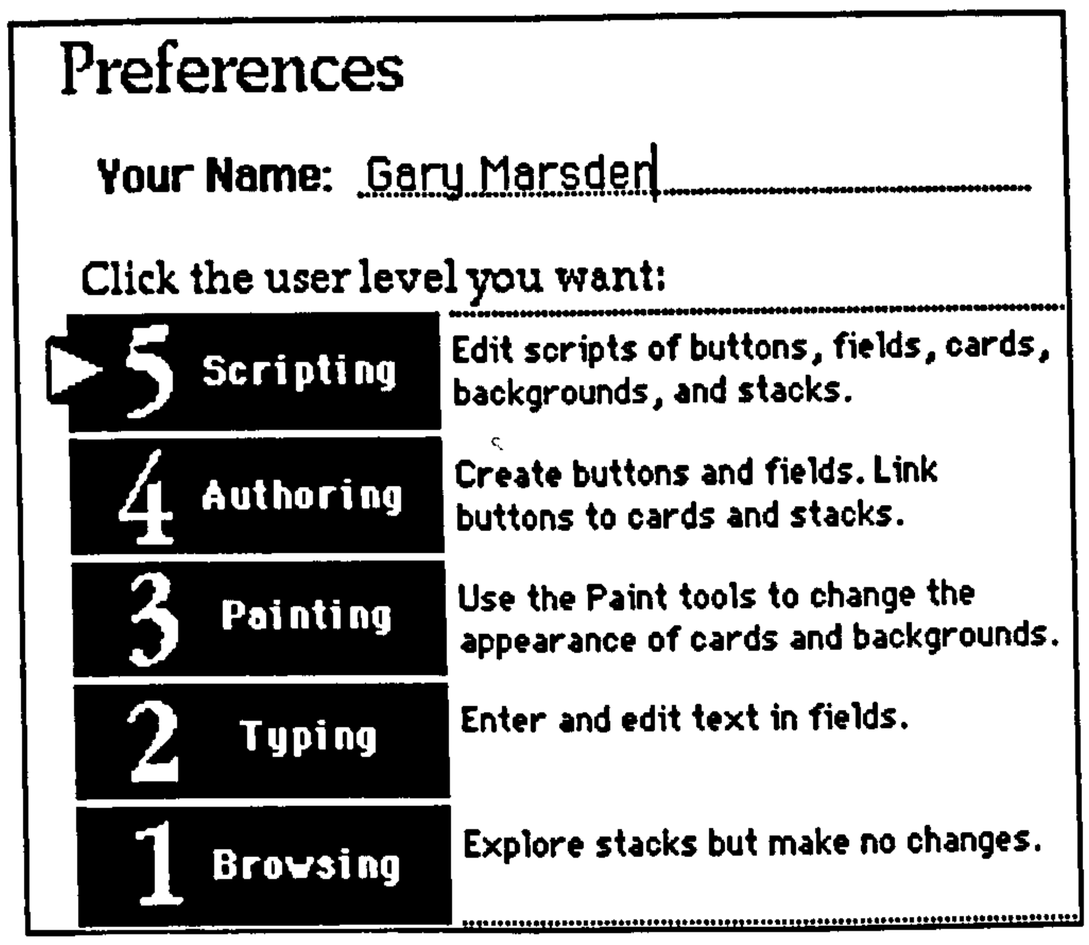


Figure 2.7: Screenshot of the HyperCard user level configuration screen

End user customisation

Most commercial applications permit some degree of user customisation — this can range from changing background colour in Netscape through to macro languages in spreadsheets. As RAD tools provide support for interface modification, it is possible that some of the functionality of the RAD tool could be built into the final application.

In fact, this is exactly the model HyperCard is built on. Within the system is a **user-level** property (Figure 2.7) which can be set to any of the following values with these consequences:

1. **Browsing:** Navigation and use of basic "File" and "Edit" menus
2. **Typing:** As above plus entry into data fields
3. **Painting:** As above plus use of painting tools
4. **Authoring:** As above and access to all tools and objects, but no scripting
5. **Scripting:** Full access and scripting control of objects

The programmer can therefore quite easily allow the end user some customisation of the application.

Visual Basic, however, by having distinct design and execution environments, cannot support end user customisation in this way — any customisation must be explicitly (and not easily) provided by the programmer.

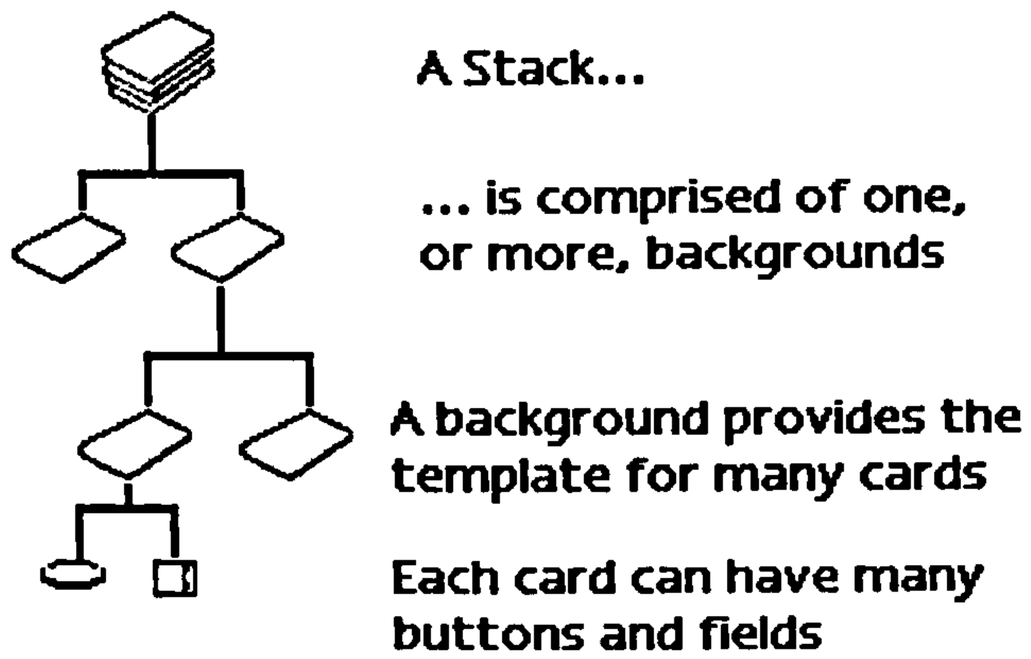


Figure 2.8: HyperCard object hierarchy

2.6.2 Language

Structure

Both HyperCard and Visual Basic are predominantly event based, object based languages. Rather than having code in one continuous sequence representing the flow of control, event based languages have code parcelled into various event handlers. Each event handler specifies which code to execute should a given event occur (e.g. **when** user presses mouse button **do** beep.) Therefore, code associated with controlling events happening to a given object is often encapsulated with that object. Because both systems are object oriented, they support the encapsulation of these event handlers with the properties of particular widgets.

HyperCard follows this method zealously, forcing every line of code to be associated with some object. To view code, the programmer must first select the relevant object and then ask to view the object's code. HyperCard has no notion of class, but does have five standard types of object (of which there can be multiple instantiations). Objects are *stack, background, card, field* and *button*, arranged hierarchically as in Figure 2.8. These objects can communicate by message passing amongst themselves, messages passing by default up the hierarchy, whenever an object receives a message it has no handler for. Because HyperTalk has no notion of classes, programmers cannot create their own class of objects, although, since version 2.2, they can create a "family" of radio buttons with a common behaviour. Programmers can create their own custom event handlers, procedures and functions.

Like HyperCard, Visual Basic has no classes, but provides various objects. It is more lax about its encapsulation, event handlers for particular widgets being stored in a common *form* file which is also used to store attribute variables of those controls. Visual Basic makes a distinction by providing code modules (i.e. invisible forms which cannot have widgets attached to them) for general purpose functions and global variable declarations. As each

object is created, event handler stubs are created in which users may place their code. This relieves users from the burden of checking which events an object can and cannot respond to. Visual Basic does not make a distinction between event procedures and general purpose procedures and hence does not use the "message" metaphor used in HyperTalk. Again, because there are no classes, users cannot create their own objects, but Visual Basic provides a form of aggregation using a "Control Array." A control array contains homogeneous objects for which common event handlers can be specified. Finally, Visual Basic has a third type of file, namely the "project" file. This file lists all the files associated with a given project, including all the required VBXs.

Neither system has any form of explicit inheritance for interface objects. The only form of inheritance is through delegation [Hor88], [Ste87].

Language syntax

As studies by Green [Gre77] have shown, syntax is a large factor in the comprehensibility of a programming language. Heeding this research, the designers of HyperTalk gave their language a very verbose syntax based on English. Whilst this may be more endearing to novice programmers, it may alienate those already familiar with an established programming language and is prone to the ambiguities discussed in Figure 1.4.

Visual Basic (as the name implies) is based on the programming language BASIC. Although no quantitative data could be found (perhaps because it is a foregone conclusion), as BASIC is part of the DOS / Windows operating system, BASIC must exist on more personal computers than any other language, making it the language most novices are likely to have learnt. Rather than using a completely new, alien language, Microsoft hopes to exploit users' knowledge of BASIC to promote familiarity with the Visual Basic system. Once learnt, users can transfer their skills from Visual Basic programming into other BASIC dialects. Although elements of HyperTalk appear in other systems (e.g. Macromedia Director [Mac97b]), there are no other tools which explicitly use HyperTalk and the transfer is less direct.

Data types

When attempting to classify different forms of data typing, the terms "weak" and "strong" provide little help in clarifying the situation. In order to improve the granularity of classification, Pressman's [Pre87] classification will be used. This consists of five levels defined as follows:

- **Level 0:** Typeless
- **Level 1:** Automatic type coercion
- **Level 2:** Mixed mode (like level 1, but coercion only applied to "similar" types, such as reals and integers)
- **Level 3:** Pseudostrong (like strong typing, but with a single weakness)
- **Level 4:** Strong

Visual Basic has the usual BASIC "string" type, but also includes the "long", "int" etc. types of C. Typing is strict (level 3) and execution will halt when a type error occurs. Type checking can be relaxed by the use of variant variables, which may hold values of any type. Combining variants in, say, an addition will only work, however, if they are currently both of the same type. Types are explicitly declared, although variant types are inferred at run time.

HyperTalk has two basic types, i.e. "string" and "number", and has a more relaxed attitude to typing and will attempt automatic coercion (Pressman's level 1). For instance, adding a string value to a numeric value will produce a concatenated string, not an error, as in Visual Basic. Adding two numbers will always produce a number. When the translation rules fail, however, HyperTalk must also resort to reporting the error.

Because of the interpreted nature of both systems, binding is dynamic.

Data structures

HyperCard has no data structures *per se*, but it is possible to simulate them by exploiting HyperTalk's string manipulation commands. For instance, data items separated by spaces in a string variable may be individually accessed using the "word" command. Obviously the user is forced to maintain the integrity of data structures constructed in this manner.

The Visual Basic type system owes a lot to C. This has much to do with the way in which Microsoft market Visual Basic. Firstly, it is expected to fill the roll of a programming system for novice programmers and secondly, it is used as an interface development tool for experienced programmers. If experienced programmers are to successfully link their C routines into Visual Basic, then it must provide support for C data structures. Consequently, besides the traditional Basic fixed length array, Visual Basic also supports dynamic arrays and a simplified notion of records — simplified in that there are records and no pointers, so records are "linked" using dynamic arrays.

Data persistency

Persistent systems, such as Apple's Newton preserve the state of a system, without the user needing to perform explicit "save" and "load" commands. For example, if a persistent system was being used to edit a file, and the user quit and then restarted the system some time later, the file would reappear in exactly the same condition.

HyperCard stacks are persistent, but require (programmable) explicit garbage collection in the form of the "Compact Stack" command. HyperTalk, however, does include file manipulation commands mainly to import and export data files with other applications.

Visual Basic has no form of persistence and requires the explicit saving of form, project and code files. This also means that it is the user's responsibility to program their own "save" routines, should they require the retention of any data values between executions.

Multimedia data types

As these tools are designed to create graphical user interfaces on multimedia computers, they must manipulate a wide variety of data. Interestingly, neither HyperTalk nor Visual Basic support these as primary data types, but rather provide specialist widgets for containing multimedia data such as images or sounds. As shall be seen, the support found in these languages is tied very closely to the hardware on which they execute.

The Macintosh was originally designed as a monochrome machine, running an overworked 68000 processor. As a consequence, the original Macintosh toolbox included only very simple, black and white widgets, which are still present in the most recent version of the operating system (currently 7.6). Although HyperCard 2.2 claims to be coloured, this is provided by means of an XCMD kludged into the system (for instance, the environment paint tools cannot paint in colour). Conversely, Macintosh computers have always had advanced sound capabilities and this is reflected by the ease with which sound can be incorporated into HyperCard stacks. A more recent innovation in the Macintosh operating system is QuickTime, which provides support for applications such as HyperCard to play video clips, again, using widgets.

Again, the capabilities of Visual Basic reflect the machine's capabilities, rather than the probable wishes of the system's designers. Visual Basic has very little ability to produce sound, as most early PCs were not equipped with a sound card. It is much harder to play video clips as Windows has no definitive compression standard. What Visual Basic incorporates very successfully, however, is colour. Every control has its own colour settings which can be

used to create colourful interfaces.

Naming Conventions

Declaration of variables in HyperTalk is entirely implicit, with the variable type being inferred. To differentiate a global variable from a local redeclaration, it is preceded by the keyword `global`. Visual Basic declarations can be implicit or explicit using the `Dim` keyword (explicit declaration can be enforced). Unlike older versions of BASIC, Visual Basic does not enforce the rule that string variables must terminate with a "$" symbol. Again, to identify global variables, a `global` keyword is provided.

Objects (such as fields and buttons) are automatically named by each system. Within HyperCard, each object is given a unique identification number, which should remain unique over time as no number is recycled. Objects can also be referred to using a name (a string allocated by the programmer) or a number (allocated by the system dependent on location on a particular card). Visual Basic object names are comprised of a text and numeric part — the text part is a description of the widget and the number is the n'th occurrence of that particular widget on that particular form (e.g. the third text box on a form will be given the name "Text3"). Programmers are free to alter these names and can refer to objects on other forms by preceding the object's name with the form name and a dot (e.g. "Form1.Text3").

HyperCard also has some unique forms of referencing, namely:

- **it:** This always refers to the last result (such as the return from a function call) encountered by the system.
- **me:** This allows button or field objects to refer to themselves, provided "me" is used within the event handler of the object (it would not work for an object to call a function located in another object which contains "me").
- **this:** Allows objects to refer to the card, background or stack on which it is located.

Error handling

Although both systems claim to be compiled, only a rudimentary amount of compilation can actually be achieved, due to the highly interactive nature of the programs they create. This, of course, results in more execution errors than would occur with a compiled language.

As the HyperCard editor has no syntax checking, it is possible to encounter syntax errors

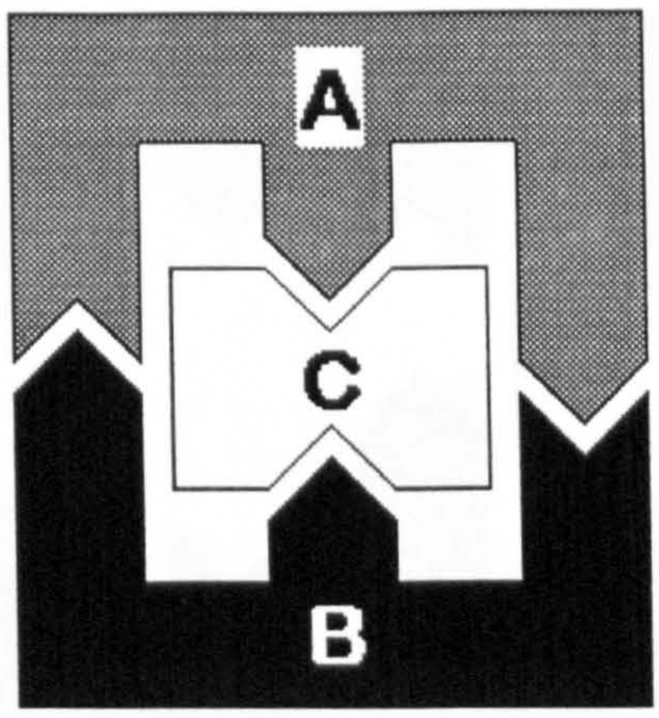


Figure 2.9: Sample component integration diagram

at run time whilst Visual Basic can eradicate these before the programmer exits the design environment.

Both systems will halt when encountering an error during execution causing the programmer to be presented with three options: "Debug", "Halt" and "Edit". Selecting "Halt" in either system will halt execution; in Visual Basic, the programmer is then returned to the design environment. The "Edit" option in both systems will return the programmer to the script editor. If the programmer successfully corrects the code, execution will continue, otherwise execution terminates. In HyperCard, the effects of the execution before the erroneous line is encountered will be preserved; Visual Basic, however, will lose the state when returned to the design environment. Selecting the "Debug" option has the same effect as the "Edit" option, except that the run-time debug tools are made available.

Integration

To aid comparison, we shall create a visual representation of the various degrees of integration as in Figure 2.9. The closeness of two leftmost legs represents the influence B exerts over A (a wide gap between the legs would show that there is little integration of these two elements). Similarly the rightmost legs represent the degree of influence A exerts over B. The middle legs represent the degree of control A and B exert over a third, passive, element C.

Finally, this diagram can be used to visualise the integration within HyperCard and Visual Basic as in Figure 2.10.

The diagram shows that the Visual Basic environment provides very good language support, yet the language cannot control and does not inherit many of the features of the environment. Integration of both language and environment with the toolkit is very high.

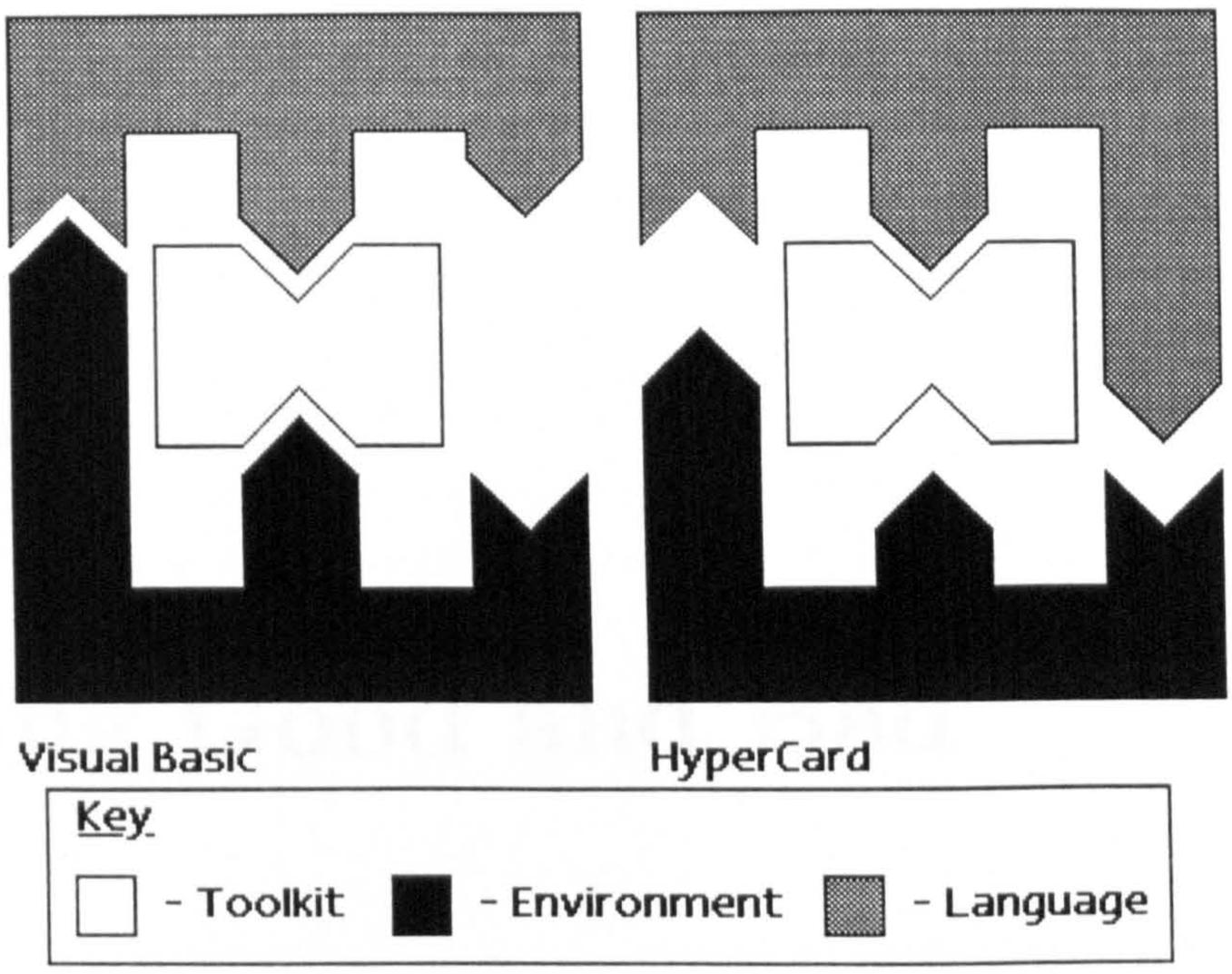


Figure 2.10: Integration diagrams for Visual Basic and HyperCard

HyperCard's environment, alternatively, provides only limited support for HyperTalk. On the other hand, much of the HyperCard environment is accessible and configurable from within HyperTalk. Although HyperTalk integrates well with the toolkit, its environment does not.

2.7 Summary

Having defined exactly what a RAD tool is and examined the attributes of two leading systems, it now remains to examine these attributes to see if they best support the novice programmer.

Chapter 3

Defining Good and Bad

3.1 Introduction

In the previous chapter, a comparison was made of the features found in HyperCard and Visual Basic. What still remains to be determined is which of those features are "good" (and "bad") for a novice programming language. Of course, before attempting this, it is necessary to define "good" and "bad." This is not as straightforward as might reasonably be expected and we must re-apply research conducted on other (non-RAD) systems and re-interpret it for the RAD paradigm.

3.2 Observations from UIMS research

3.2.1 Some badness

The majority of analysis into what constitutes "good" and "bad" for interface programming systems has been carried out by Brad Myers. The following is a list of what he considers to be "bad" attributes, as he reports in [Mye92b]:

1. Lack of appropriate I/O mechanisms
2. Lack of support for multiprocessing
3. Inefficient Object Oriented (OO) mechanisms

4. Poor support for rapid prototyping

5. Inappropriate representations of concepts

6. Lack of new features in the language

It is doubtful anyone would argue that these are undesirable features. On their own, however, they provide little help in guidance of better ways to design new languages. (How, for instance, is one supposed to make sure the language has efficient OO mechanisms?) Each item in the list is really a symptom of some deeper malady: one might add to the list indefinitely examples such as "ill-suited syntax" or "inappropriate data types." If improvements are to be made, then a more generalised, higher level rule must be found to guide the design of any new system.

By searching for a more fundamental rule from these generalised rules and by applying them to the feature lists from the previous chapter, we can hopefully create a comprehensive list of specific instances of "badness" (and also "goodness") for RAD systems.

Certainly one root cause for the first five symptoms in the above list is the adoption of a third generation procedural language as the basis for most RAD tools. Examining each of the five points in turn would seem to support this conclusion:

- **Lack of appropriate IO mechanisms:** As these languages were designed to receive input and create output as streams of text, it is little wonder they are not suited to creating GUIs which handle asynchronous input and output in both text and graphics.

- **Lack of support for multiprocessing:** As third generation languages were created to run on quite limited hardware, multiprocessing was not possible and these languages were designed to support a single "flow of control." Although there does not yet exist an agreed "best" way to implement multiprocessing, techniques such as event programming seem to present a more workable alternative.

- **Inefficient Object Oriented mechanisms:** There is strong evidence [RA90] that object orientation is a good paradigm for implementing GUIs. Using a non object oriented language is missing an ideal opportunity to reduce conceptual complexity. (Object "enhanced" versions of older languages may suffer from legacy conceptual baggage, but go some way to overcoming limitations in the language model.)

- **Poor support for rapid prototyping:** Third generation languages were designed to support traditional software development cycles with long specification and design periods before any implementation was attempted. This led to languages with long

compile and link cycles — hardly appropriate for new rapid prototyping methodologies or impatient novice programmers [1].

- **Inappropriate representations of concepts:** With this point, Myers is trying to communicate the disparity between the textual representation of the program and its appearance on screen. Again, third generation languages were designed to support the functionality of the application, and lent themselves to be broken into modules, each module corresponding to an individual task in providing the overall application. As user interface code now accounts for approximately 50% of the code for most applications and, correspondingly, 50% of the effort in creating an application, the language constructs should do more to harmonise application and interface concepts. (In [Raa87], Rhyne states that "researchers generally agree that high-quality user interfaces do not hide the application semantics from the user").

The final symptom from Myers' list is lack of new features adopted into languages. This can largely be attributed to the type of research and commercial development being invested in this class of tool. Firstly, the commercial sector has no desire to fundamentally alter the underlying language and alienate programmers in other languages (a reason explicitly stated by Sun in their choice of C syntax for their new programming language, Java). Secondly, the research community, who might contribute, are divided into language designers (who, as Myers notes, are not concerned about the higher level aspects of languages) and HCI specialists, who are more interested in developing graphical environments than worrying about creating a completely new type of programming language.

So, what has been learnt by looking a little deeper, behind the symptoms?

It is not enough to use a language with which programmers are familiar, or which was once considered to be a good novice programmer's language. The attributes which may have made it a successful tool in the past may no longer hold true for use as a RAD language. Before it can be used, thought should be given to the likelihood that it will produce any of Myers' "badness" features listed above.

As was previously discussed, this list is by no means exhaustive and it would seem that the best way to find a RAD programming language is to design a new one, optimised to the concepts required for interface programming. This way, even if new problems are created, needless legacy problems will have been eradicated.

---

[1] The distinction between compiled and interpreted languages is blurring with "compile on demand" technologies.

Having defined what is "bad" and how we might remove it by eradicating reliance on old language technology, it remains to look at the "good" and hopefully identify where that comes from.

3.2.2 Some goodness

Again, Myers, in [Mye91], lists the following as being desirable attributes for GUI programming systems: that they

1. help *design* the interface given the users' tasks
2. help implement the interface given a specification of the design
3. allow the user to rapidly investigate different designs
4. allow non-programmers to implement and design user interfaces
5. allow the end-user to customise the interface

So what can be generalised from these points? The first four design suggestions can be effectively summarised as follows: *the semantic concepts of the language should reflect the problem domain.* Hardly a revolutionary statement, yet one which has guided the design of all fourth generation languages. These problems all stem from the low-level focus of current toolkits and languages — they approach interface creation as a task in manipulating screen areas and widgets states, *not* supporting the interface designer in mapping between widgets and their use in the final application. (A view supported by others such as Foley, who also cite this as a motivation for their work [Fol86].)

The last point is interesting and supporting this properly has (we believe) huge implications for the overall design of the programming system. The reasons for this will be explored later but, for now, we shall accept that this is, self evidently, a valid design goal.

These symptoms and goals identified by Myers were found in the *language* component of interface creation systems. What problems, then, might one find in the interface *building environment*?

3.2.3 The Environment

The usual caveats for designing any form of direct manipulation interface also apply to the creation of interface builders. Papers such as [MAS91] provide lists of absolutes and guidelines

for designing general classes of direct manipulation interfaces. This class of application, however, presents some unique problems.

Execute and design modes

One problem unique to RADs and UIMSs is the existence of *design* and *execute* modes mentioned in the previous chapter. Direct manipulation interfaces exploit the user's knowledge of real world objects by using analogous, or metaphorical, on-screen representations of these objects. Provided the analogy is strong, the user can interact with the software without any explicit training. Problems occur, however, when the metaphor falls apart and the screen object behaves differently than the user had anticipated. Classic examples, such as the Macintosh trash can ejecting a disk, can seriously undermine a user's faith in what they have learnt to date about a system.

This is especially true when a user wants to start customising and writing their own interfaces. Novice programmers who use an interface builder for the first time encounter a unique problem. Rather than the expected response of highlighting a box, clicking a check button in the interface builder environment causes a check box property dialog to appear! The analogy between real world and on screen objects can no longer be maintained. If the programmer is to successfully alter and create interfaces, they must learn that widgets have an "execution" behaviour (with which they are already familiar) and a completely new "design" behaviour. It is this new design behaviour they must master in order to control the execute behaviour. Of course, this is really just another manifestation of a "mode" problem (similar to those identified in [Mon86]). The uniqueness of this application, however, has produced a variety of solutions, none of which could be termed optimal [SUC92]. In fact, these two modes are so fundamental to UIMS that most systems do not try to disguise them, but confess to having separate design and execution modes. Other systems, which do attempt to hide them, often leave the novice programmer confused about when they are editing and when they are executing [HS95]. For an example of this poor feedback, see Figure 3.1.

Integration of environment and language

The nature of RAD tools, having a visual environment to draw the interface and a separate language to define the application behind that interface, creates some unique learning problems. In contemplating this, Dertouzos [Der92] observes:

> ...such prototyping software stops being useful at exactly the most interesting point of the programming process...what is needed instead is the ability to pro-

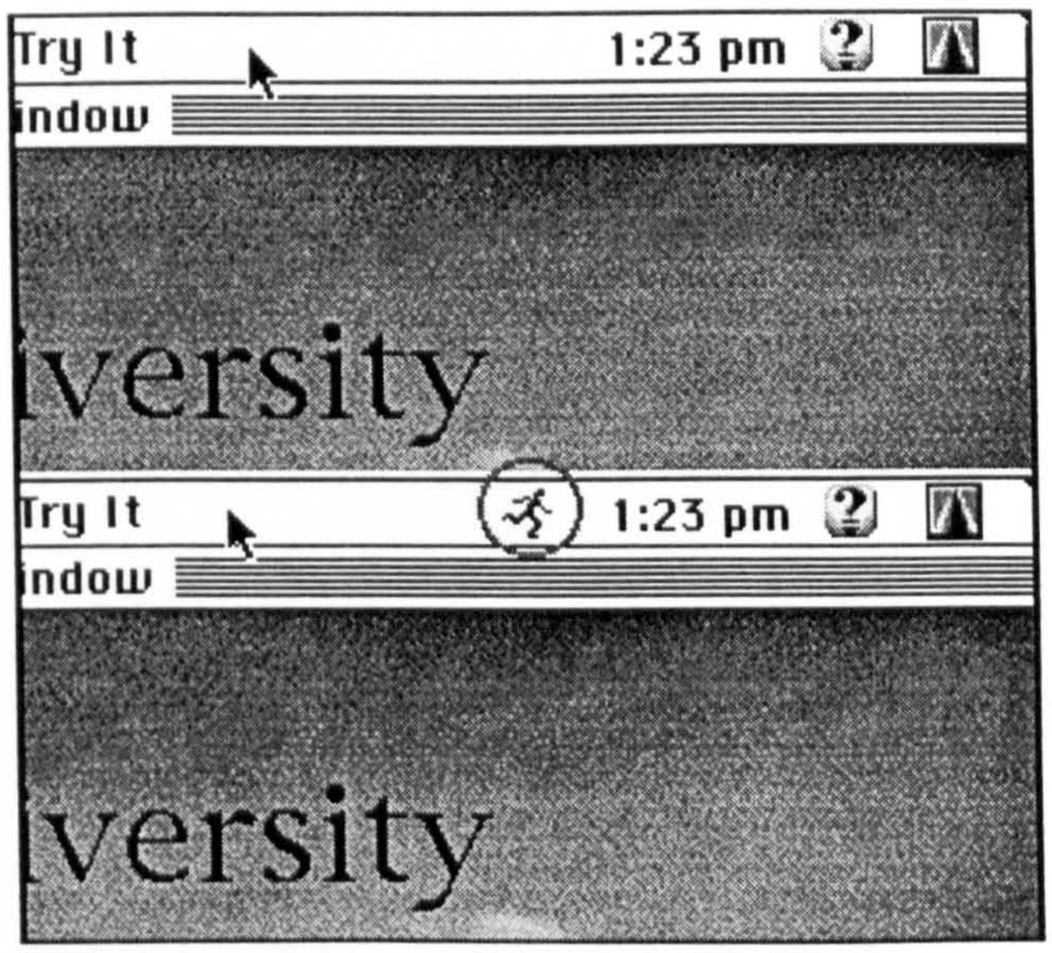


Figure 3.1: The upper image is of a Macintosh menu bar for AuthorWare in design mode; the lower is AuthorWare in execute mode. Apart from the small "Running man" (here highlighted by a circle) there is no other feedback to the user.

ceed smoothly from prototyping the user interface.

Once the novice programmer has mastered the interface builder, they can construct an interface to any application they might choose to implement. Before they can implement that application's functionality, however, they must learn an entire programming language. This can leave the novice disappointed or discouraged to the point that they abandon the implementation. An ideal tool would somehow combine the immediacy of the direct manipulation interface builder with an understanding of how to create the application's functionality. Figures 3.2 and 3.3 attempt to represent the various learning stages encountered by the novice programmer.

Although hardly a rigorous, or even a real curve, the notion of "learning curve" serves as way of visualising the stages of learning a programming language. An ideal tool would provide continuous return for the amount of effort invested in learning that tool. RAD systems, such as HyperCard and Visual Basic, however, have discrete levels, or *kinks*, in their learning curves.

From the origin to stage (A), the programmer can use the environment to place widgets and tailor the interface. From (A) to (B), however, the programmer makes little, or no, progress as they spend their time learning the concepts of a programming language. From (B) to (C) the programmer can use the language to implement increasingly complex applications. After this point, the scripting language is no longer able to provide the functionality required by the programmer. From (C) to (D) the programmer must learn another language to write

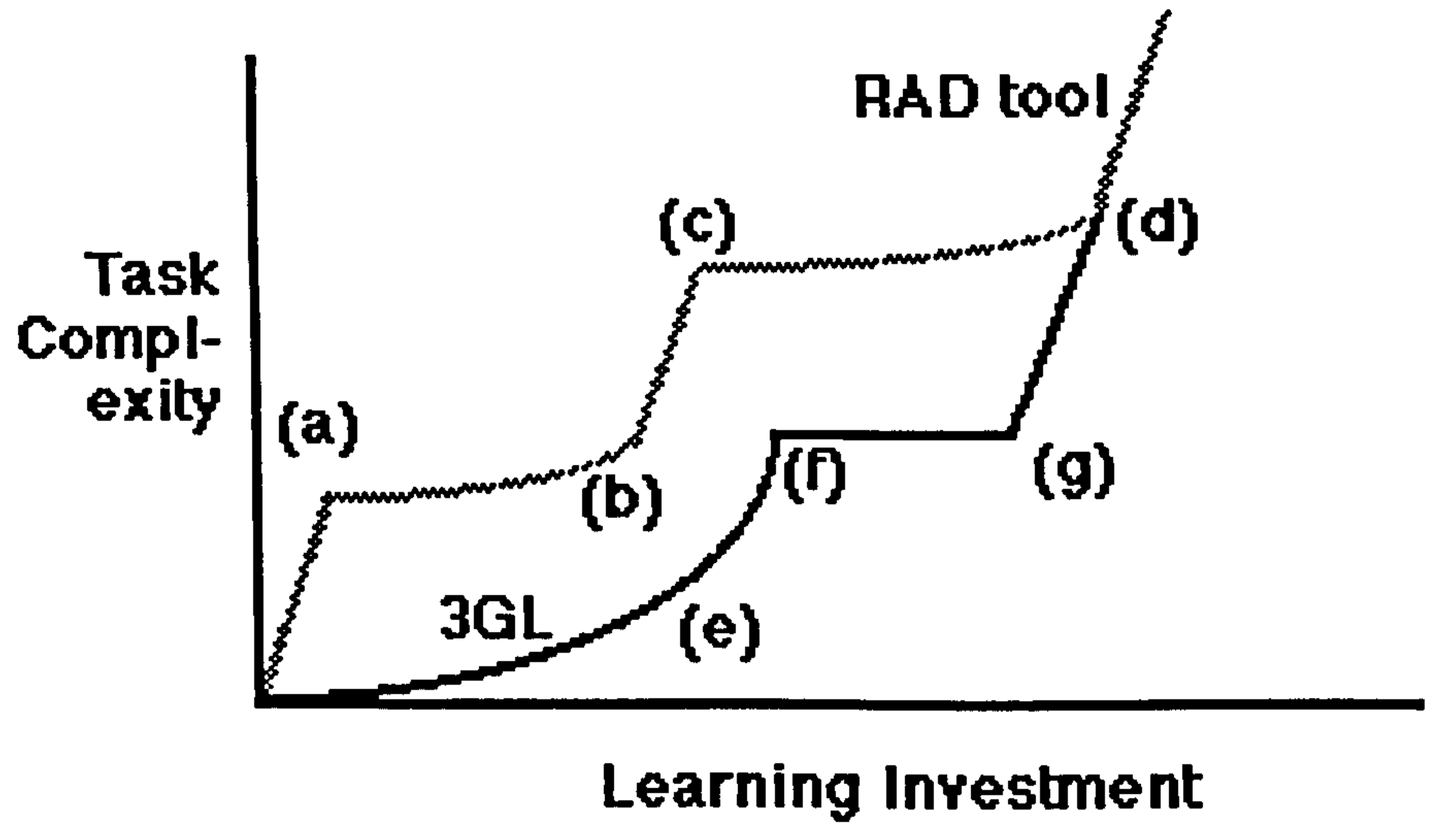


Figure 3.2:

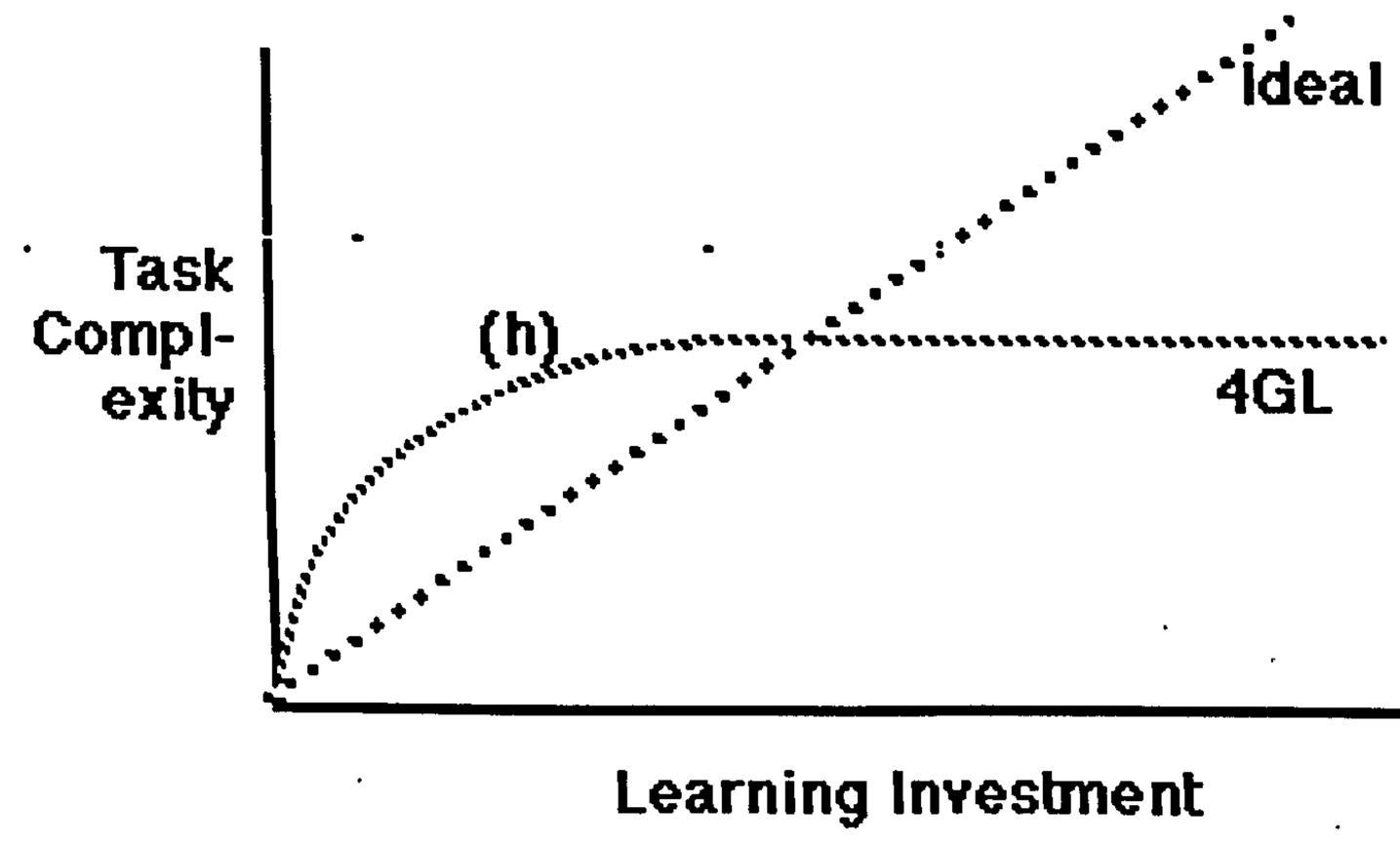


Figure 3.3:

extensions (VBX, DLL, XCMD etc.) to overcome the limitations in the script language. After (D) the programmer is free to implement any application.

Included in the graph is a third generation language (3GL) which does not provide the same return initially, period (E), but eventually enables the programmer to implement any application without a GUI. If the programmer requires a GUI, then the period (F) to (G) must be spent to learn a toolkit and its interface. After (G) the programmer is free to implement any application.

The remaining line on the graph represents a fourth generation language (4GL). This provides high initial return, but after some point, (H), does not support more complex applications.

By taking a third generation language and combining it with an interface builder the result is, at the same time, something which is both more and less than the sum of its parts. The synergy created by this mix manifests itself in an obvious enthusiasm for the tool and an empowerment of the novice programmer. Also created is a system which has superfluous concepts, a far from seamless join between language and environment and systems which typically tie themselves to one type of hardware.

3.2.4 Summary

Distilling the preceding discussion into a list of summary points provides the following, which will be used to ascertain the appropriateness of the language features examined in the previous chapter:

- The language should have no superfluous features inherited from ancestors.
- The system should be orientated to the domain of the problem.
- The system should permit code structure to reflect interface structure (if desired): to support prototyping, therefore, the system should not require the code structure to be decided before implementation begins.
- The building environment should integrate smoothly with the underlying language and should not result in redundant concepts.
- The system should allow the programmer to migrate smoothly from novice to expert.

By applying these principles to the features of Visual Basic and HyperCard from the previous chapter, we can now generate both good and bad "symptoms" specific to RAD tools.

3.3 Searching for Good and Bad in RAD

3.3.1 Environment

Modes of interaction

There is, as yet, no consensus on how best to represent design and execute modes (this problem also occurs in spoken language where it causes ambiguity, being termed "use - mention" [SUC92]) resulting in a diversity of manifestations.

Visual Basic supports design and execute modes which are clearly identifiable. By adopting this approach, Visual Basic does not support rapid prototyping as well as it might because time is wasted in switching between these environments. There is also the introduction of mode problems and extra cognitive load on the novice programmer. This is no doubt inherited from previous versions of BASIC and hence could be considered an inherited, superfluous behaviour.

HyperCard abandons the notion of a traditional compile and execute cycle and adopts a "tools" approach, albeit a rather strange variation. Rather than taking the semantically driven approach of providing an "execute" tool and a "modify" tool, HyperCard has two versions of the "modify" tool, namely the "field" and "button" tool. This is obviously a legacy from when HyperCard only supported these two types of widget and it causes classification problems as HyperCard's widget library is extended. This, of course, results from viewing the problem at too low a level and could have been solved had the designers taken a more semantic view of the problem.

Also, within HyperCard, it is impossible to modify some parts of the interface whilst others are executing[2]. In [SUC92] Smith calls this a "system wide tool" as opposed to the "per object" tool, which does allow simultaneous modification and execution of the interface. The "per object" tool is more in keeping with the novice's experience of real world objects (such as buttons) to which the screen objects are supposedly analogous.

Although HyperCard clearly identifies the change in mode with the change of cursor (this is shown to be an effective form of feedback in [Gav91]), the affordances shown when a field or button is selected for modification are different to those used in other Macintosh applications. Consequently, as Nielsen discovered in [Nie89], this caused confusion for novice users. Clearly HyperCard suffers from an integration problem with the toolkit in this respect.

---

[2]This is broadly true, but the use of the "idle" handler in a script allows simultaneous execution and editing.

In summary:

- Ideally, there should not be separate design and execute modes.
- The integration of modes should not confuse the programmer about when they are designing and when they are executing.
- The behaviour of objects should not conflict with the programmer's previous experience of using them in applications and should, so far as is possible, behave like their real world counterparts.

Code translation

Both systems use "compile on demand," which is ideal for supporting novice users exploring the tool and iteratively designing applications. Even the original versions of BASIC were interpreted and the benefits of interpreted (or "compile on demand") systems prevented the language designers from adopting a more traditional compiled language approach. This means, for example, that:

- The program does not need to be completed in its entirety before it can be executed.
- The user is not confronted with the compiler errors for the entire program, but just the sub program which is being compiled.

For the criteria of a RAD tool, this method of code translation is probably optimal (although any improvements in execution efficiency are always welcome!)

These systems, by their nature, however, will never produce small and efficient code. Their reliance on a run time system will inevitably mean that part of that run time system will need to be distributed with the application. Obviously it is better to allow the programmer to specify whether this functionality should be compiled into the application or whether the run time support can be provided as a separate library or application.

In summary:

- Compile cycles should be as short as possible (or at least incremental).
- It should be possible to compile a final distribution version of the software.

Extendible widget set

Both systems can be extended with new widgets. Extensions to the Visual Basic widget set appear in the palette along with the standard widgets. These may be inspected with the "property browser" and treated as standard widgets. Visual Basic, however, does not support the direct invocation of methods in VBX controls — methods are invoked as a side effect of setting a variable value! For example, the "Graph" control will redraw a graph when its "DrawMode" property is set to 4. Software engineers have known for a long time that side effects are poor practice and there is little excuse for them to appear in a modern programming language. This is, perhaps, the result of adopting a language (BASIC) which was not designed for structured or object-oriented programming.

The HyperCard XCMD and XFCN protocols, as the names suggest, were intended to add new functions and commands to HyperTalk. As XCMDs can directly call the toolkit, they can cause widgets to be displayed and controlled. This means that all interaction with widgets is by method invocation — a much better abstraction than that used by Visual Basic. However, the HyperCard environment cannot display the interface to the widget, leaving the programmer to remember the calls from the XCMD documentation. This results, again, from the low level view taken by the HyperCard designers, who did not envisage the need for new forms of widget. This is in stark contrast to the "field" and "button" widgets of HyperCard for which it is possible to view (some of) the properties of widgets and display all the methods to which they can respond. XFCN and XCMD widgets are, unlike VBXs, *not* integrated into the environment in the same way that the standard widgets are.

Clearly the ideal solution lies with a hybrid of the two systems. The environment should display the interface to the widget but interaction with the widget should not rely on side effects. Any widgets added to the system should be treated in the same way as widgets supplied with the system.

Of course, the ideal system would support the creation of new classes of widget. Until toolkits are built at a higher level of abstraction (as may yet happen with Java) this is unlikely to be possible.

In summary:

- Systems must provide for the inclusion of new widgets and support these on the same level as the standard distribution widgets.
- Control of these widgets should be achieved using alteration of instance variables *and* invocation of methods — not exclusively one or the other.

- The interface to these widgets should be visible, using some form of object browser.

Platform support

It is interesting that neither of these systems has been directly ported to another platform. Both were created as an attempt to hide the underlying complexity of the toolkit and allow novice programmers to program. Consequently, both systems are tied very closely to the toolkits of the host operating system. The subsequent development of each system, then, is largely driven by developments in the features provided by the operating system. Neither is it in the commercial interests of either manufacturer to provide cross platform support because, in itself, the desire to use a particular tool may influence the purchaser's choice of operating system.

By following the development of the operating system, these languages will inevitably become more complex. The accompanying documentation to new releases of these systems always starts by listing the new features added since the previous release — rarely do these additions make the tool easier to use. Commercial developers will listen to user feedback (from questionnaires and news groups), but this type of user input to development will lead to the conclusion that the user community requires more features. For example, if this is the first system a programmer has used, they will not be aware of different programming paradigms and hence cannot suggest this type of improvement to the manufacturer — they will only suggest "patch," low-level improvements to overcome particular problems they have encountered. The following quote from [LWSS80] would seem to apply to the development of the RAD tool:

> In the development of any technology there is always a tendency to lose sight of the basic problems that stimulated its introduction. After the first flush of success, interest naturally tends to focus on the technology itself and the problem it presents.

The evidence for this is shown by Green in [Gre90a] which was published in 1990, who thought that HyperCard was a good beginning and was looking forward to Apple cleaning up some of the conceptual problems "in the next attempt." Seven years later, we are still waiting.

The success of Java has shown that there is a need for a non platform specific GUI programming language. Java, however, adopts a lowest common denominator approach and can often produce "clunky," unattractive interfaces. Another approach is to provide a rich, platform independent toolkit (an approach taken with XVT [XVT97]) which allows the programmer

to create the interface using an abstract tookit. This solution, however, relies on the entire toolkit being implemented for that operating system. If the language allowed some higher level specification of the functionality of the interface (not specifying particular widgets) then it would be possible to exploit the richness of any underlying toolkit.

Therefore, by providing a higher level language which specifies interaction requirements, rather than specific widgets, it would be possible to provide a platform independent system which is able to make best use of the hardware / operating system it runs on. Also, language developers will be able to continue to add new features and widgets without directly increasing the complexity of the language.

Sadly, the platform-centric nature of UIMS (as they were then termed) was noted by Myers [Mye87] some ten years ago. Again, we are still waiting for the situation to change — the interface abstraction is not orientated to the domain of the problem, but to the machine domain.

In summary:

- The language should not rely on specific hardware or operating system features.
- The language should view widgets not simply as what they are (i.e. some form of code object) but what they do (i.e. select values, invoke routines etc.).

Program Editors

The editors of both systems are syntax aware, which permits:

- Detection of syntax errors
- Automatic indentation
- Statement colouring

Work by Green [Gre90b] has shown that these facilities contribute significantly to program comprehension, but these systems fail to provide support on a higher level, which could aid programmers with, say, command recall and help reduce errors in code entry. Again the improvements currently provided are at a low level, merely embellishing the type of product that has gone before. Instead, it should be possible to provide a more domain orientated editor which better supports the novice programmer.

In summary:

- The editor should be aware of the language and reduce the possibility of entering incorrect lines of code.
- Editors should provide facilities to aid program comprehension.

End user customisation

The "UserLevel" abstraction of HyperCard works well in providing end user customisation as so much is provided "for free." Potentially, it makes available to the end user all the tools that were available to an application's author. For example, if the author wants to permit the end user to edit the icons of an interface, they do not need to write their own icon editor, they can simply make that part of HyperCard accessible from their application — in other words, there is good integration between environment and language. Visual Basic programmers, however, must replicate any functionality of the build environment they wish to perform in their application. By adhering to the more traditional approach of the compile-execute cycle, Visual Basic increases the burden on the programmer and potentially lowers the quality of application delivered to the end user.

Although HyperCard's solution is useful, it does not provide a fine enough grain to the level of customisation. Quite often, the programmer will discover that the "UserLevel" mechanisim is insufficient and they must resort to programming levels of customisation explicitly (as with Visual Basic). For example, trivial properties, such as button labels, cannot be altered until the user is given level 4 clearence.

Also, the correlation between interface structure and the structure of the underlying application code has a large part to play in facilitating the customisation process. If, as is the case with object-oriented toolkits, the behaviour and attributes of interface widgets are encapsulated with that widget, the potential for end user customisation is greatly heightened.

In summary:

- It should be possible to use elements of the interface builder in the final application.
- There should be some hierarchy imposed on the attributes of each widget.
- The structure of the interface should reflect the structure of the program.

3.3.2 Language

Structure

Currently, there would seem to be no better paradigm than object-orientation for the creation of this type of interface [CH86]. The notion of a graphical interface object having behaviour and attributes is broadly consistent with a novice's view of a physical object in a real world interface (Thimbleby terms this a *reflexive interaction paradigm* [Thi90], where the language paradigm is successfully carried through to the interface). The implementation of object orientation which HyperCard and Visual Basic present, however, is very different to that found in full object oriented languages such as SmallTalk [Gro81] and Sather [Gro97a].

One useful implementation feature of both systems is the way in which objects can be cut, copied and pasted as easily as text in a word processor. By implementing object creation in this way, programmers do not need to worry about sending instantiation or creation methods to objects — object creation seems very natural and familiar. It could be argued that this does provide a very limited form of static inheritance. There is also evidence [RA90] that, for novice programmers, this is the best form of inheritance, as it relies on concrete object instances, not the more abstract class mechanism. However, using this method, it is not possible to create virtual objects, or objects which are pointers to other objects.

The lack of classes in both systems is typical of the type of plateaued learning curve problem identified previously. Whilst it is accepted that the notion of*class* is an abstract concept, harder to identify than a specific *object* instantiation, it does not mean that it should simply be removed from the language. The designers could have used a scheme whereby the novice started by using objects, being introduced to classes as their needs required. The "family" concept of HyperCard and the "Control Array" of Visual Basic are little more than patches which were necessary to facilitate the programming of radio buttons. This type of patching will inevitably lead to more complexity than if the languages had provided classes from their inception.

In fact it is more accurate to say that HyperTalk and the Visual Basic language are not object oriented at all — they are procedural languages frozen in an object environment. This inevitably causes problems when interacting with external objects (as the VBX and XCMD protocols have shown) because the type system has first class text and number types and cannot cope with treating objects on an equal footing.

Besides these common problems, each system has its own unique quirks.

Beginning with HyperCard, the provision of procedures and functions *as well as* messages is

confusing to the novice user as it is not immediately apparent what the difference is between these. Once again, it seems, HyperTalk has functions and procedures simply because they were found in ancestral languages.

The novice programmer must also make decisions about where to locate any subroutine.

All HyperTalk code must be contained in one of the five object types, and it is not always apparent where the most appropriate location should be. Whilst it is simple to decide where to place event handlers for a particular object, it is not clear, for instance, with which object an upper case to lower case conversion function should be associated. Having placed the code in objects, it is often difficult to find it again as HyperCard provides no way to print the scripts for all objects. (It is possible, however, for the experienced programmer to write such a script for themselves.)

Visual Basic does not suffer from these problems, event handlers merely being a specialised form of procedure. Code location problems are solved by the use of general purpose modules. Providing both forms and modules is a distinction which is not really necessary, modules being another abstraction found in ancestral languages. Instead, modules could be invisible forms or forms could be visible modules.

One unnecessary imposition enforced by Visual Basic is the explicit inclusion of required VBXs in the project file. Rather than including them automatically, an error is reported if a control type is used in a form but not included in the project file.

In summary:

- The language should provide a flexible object-oriented model.
- The language should have a clear and consistent object model with no legacy exceptions.
- The language should include both static and dynamic inheritance mechanisms.

Language syntax

Although determining what is an "ideal" syntax can often be a subjective affair, in this instance we are examining syntax for a particular set of users (novice programmers).

The English-based syntax of HyperTalk may initially seem like a plausible solution, but there are a number of problems with using a natural language as a programming language syntax. Unlike formal languages, natural languages suffer from ambiguities, allowing well formed sentences to have different meanings for programmer and computer. As current parser technology

is a long way from successfully analysing natural language, HyperTalk is based on a sub-set of English. This may mislead programmers with no experience of other programming languages, causing frustration when they use English constructs beyond the parser's capabilities.

Not only are there problems in adopting English, but HyperTalk has other syntax problems:

- The syntax is inconsistent, precluding orthogonality of constructs and making the language harder to learn. For example, `go card 5` and `go card 5 of stack "me"` are legal syntax, but `there is a card 5` is legal and `there is a card 5 of stack "me"` is not!
- HyperTalk syntax exhibits the "dangling else" problem, inexcusable for a modern programming language [WC88].

BASIC is a mature language and Visual Basic is a highly structured derivative of the original BASIC. As a result, most problems have been removed, or, at least, are well understood (not that this will help the novice).

In summary:

- The syntax should not introduce ambiguities for the sake of "friendlier" constructs.
- It is more beneficial to the novice programmer to learn a syntax which closely resembles a language they may encounter in the future.

Data types

HyperCard maintains only the number and text data types which are conceptually simpler for the novice programmer to understand than a system involving floats, ints and other forms of lower level data types. Visual Basic allows the user to start with simple 'num' and 'string' types (as in the original versions of BASIC) but also provides 'C' style types, should they be required. The type system for both languages is orientated to the problem domain, with Visual Basic supporting the programmer's transition from novice to expert.

Visual Basic is strictly typed, but this can be circumvented by the use of 'variant' variables — a variable without a type. Coercion is automatic between variables of this type but care should be taken with text boxes which hold variant values, as adding two text boxes always produces as string result. This can cause confusion:

> **For example:** If Text1 and Text2 are text boxes both containing the value 5, then Text1 + Text2 = 55. If Text1 and Text2 were variant variables, the answer would be 10. This is a little alarming. Automatic coercion is considered a bad thing by the programming language purist [Mee81], but can be excused to support the novice programmer — applying different coercion rules to identical values is inexcusable.

Again, this problem stems from viewing the external widgets at too low a level — the text widget is seen as a some sort of external interface device, not a form of visible variable holding part of the program's state.

HyperTalk is effectively typeless and because no operator is overloaded, the type of coercion problems found with Visual Basic '+' do not occur. Variable values are re-interpreted in context so as to cause a minimum of errors. However, it is impossible to enforce strict typing.

Both systems exhibit dynamic binding which, according to Meyer [Mey89] is the best type of binding for systems promoting iterative development techniques.

In summary:

- The type system should be forgiving, but scale to allow strong typing and more specific data types.
- The language should not treat semantically identical objects differently, simply because one is visible and another is not.

Data structures

The lack of data structures within HyperTalk is a puzzling omission; surely it would have been possible to provide at least static arrays. Green reports in [Gre90a] that this was a design decision for HyperCard to avoid difficult concepts — data structures were felt to be difficult and were therefore cut from the language. This lack of data structures requires programmers to resort to sleight of hand techniques, such as creating invisible fields in which to 'hide' data. As it is, this "feature" of HyperCard prevents the programmer's progression to an expert.

Visual Basic provides fixed and variable arrays and a record data type. The linking of these records using an array is probably more to do with implementation constraints (why create a new data structure when there is one available which might do) than trying to create a simple and coherent language for the programmer. If the aim was to hide from programmers the complexity of pointers, then why not adopt a list data structure as used in functional languages for this purpose?

In summary:

- It is essential that the language provides some form of dynamic data structure to prevent hacked solutions, and to introduce the programmer to the concept of proper data structures.

Data persistency

One area in which HyperCard differs greatly from Visual Basic is in data persistency. One benefit of persistent systems is in removing the distinction between files and variables within a programming language. If a file is persistent, then a programmer can conceptualise it as a permanent variable, without the need to learn file manipulation commands or concepts. Unfortunately, neither Windows nor Mac OS provide persistency at the operating system level. Individual applications must therefore provide this functionality, should it be required. As with most high level improvements, however, there is a slight performance impediment [3].

By making programs persistent, there are problems in the programmer making mistakes which are, in effect, immediately saved. To overcome this, any persistent language should enforce a strict revision control system. Sadly, it is often impossible in HyperCard to undo a change that you would rather not have made. Other persistent languages, such as Napier-88 [Gro97b] employ database techniques of rollback and commits to alleviate this problem.

In summary:

- The use of persistency means that not only is the coding overhead reduced, but it also reduces conceptual overheads and the frustration of forgetting to save files.
- The use of persistency should also be accompanied by a comprehensive "Undo" facility or some form of revision control.

Multimedia data types

The attempts to attach a widget set to a procedural language, which is essentially what HyperCard and Visual Basic are, look especially weak when considering how these systems cope with providing multimedia capabilities.

---

[3] The persistent operating system *Grasshopper* claims to be as fast as Unix, and already supports persistent Java [DHF96].

Consider the provision of graphics, for example. Within HyperCard, graphics must be drawn directly onto backgrounds or cards. Presumably these are stored internally in the stack as an attribute of the card or background, but are not accessible to the programmer as a data type. To create interactive graphics (other than icon animations) the programmer must list user mouse movements and clicks. Visual Basic provides all graphics (apart from a simple line) as a display area in an image widget. This image cannot be manipulated during run time by the Visual Basic environment. These are both low level approaches and poorly integrate the language with its environment. Other research systems, such as Tweedle [Asa87] even provide graphical objects which can have their own encoded behaviour, truly object oriented.

It is hard to see the rationale behind languages which create graphical interfaces but do not have an image data type.

In summary:

- The language should provide data types and associated operators for images, sounds and other multimedia data types.

Naming and Referencing

It is refreshing to see that Visual Basic has abandoned the idea of forcing the user to declare string variable names terminating in a "$" symbol. The use of the "." notation of Visual Basic is also a lot more concise than that of HyperCard (see the following box). By using the terms *it, me, this*, HyperTalk manages to reduce the verbosity of the syntax and produces relative addresses, making scripts more generalisable. This is a technique employed with great success in other fourth generation languages — the benefit of relative addressing for novice spreadsheet programmers is reported by Nardi [NM90].

For example, compare referencing a field on a screen in both HyperTalk and Visual Basic:
Visual Basic: `Form1.Text1`
HyperTalk: `card field "1" of card "1"`
Such cumbersome syntax can only lead to tedium and error through mis-typing.

In summary:

- One of the benefits of using a text language over a visual one is the conciseness and terseness of the representation — overly verbose syntax should, therefore, not be used.
- The use of relative addressing provides benefits in the generalisability of novices' scripts.

Error handling

By not using a compiled language, there can, of course, be no compilation errors. Visual Basic handles syntax errors within the editor, whilst HyperCard will ignore them until execution time. It is not clear which is the optimal solution: obviously HyperCard is reporting two different types of error at run time, but the programmer is not forced to finish a particular method before executing the program (as with Visual Basic).

Because of the separate design and execute environments, being interrupted during execution of a Visual Basic program has quite different effects to interrupting HyperCard execution. For example, in HyperCard, global variables will not lose their values, whereas those in Visual Basic will not be preserved. In an attempt to overcome this problem, Visual Basic will permit a semi-design mode in the middle of the execute mode whereby the programmer can edit the offending code. This can be confusing to the novice: if the error is corrected successfully, execution will continue; if the error is not corrected successfully, the program will re-enter design mode.

Obviously the removal of separate modes provides HyperCard with a conceptually sound method of interactive debugging. Also, because both languages are interpreted, it is more straightforward to provide source level debugging — especially important for the novice programmer [DT80].

In summary:

- Source level, interactive debugging should be provided and in such a way that it does not leave the programmer confused about the current mode of the system.

3.4 Summary

This chapter set out to:

1. Look at the types of problems and benefits identified by others researching interface creation tools.
2. Distil from these lists the root causes of the problems and improvements.
3. Use these root causes in the evaluation of RAD tool attributes investigated in the previous chapter.

It now remains to explore the design of new systems which incorporate the positive attributes listed above, whilst removing as much of the negative as is possible.

Chapter 4

Improving our lot

4.1 Introduction

In the previous chapter, comparisons and evaluations of GUI programming tools were made to argue what was good and bad in current RAD tools. Whilst this is useful in discovering their state of health, if RAD tools are to be improved, then ideas must be adopted from outside the immediate GUI field[1]. This shall now be undertaken, using research and ideas from programming language design and psychological studies of the programming task.

This process will first require some high level decisions about the most appropriate programming and environment paradigms. The integration of language and environment will then be considered and the ramifications of this in terms of lower level language constructs will ultimately be discussed.

4.2 Overall design

4.2.1 Re-application of third generation rules

The previous chapter discussed exclusively the opinions of specialists in RAD and other UIMS systems. To provide new directions for developments in these systems, research into third generation technology can be exploited. In terms of the history of computer science, third

[1] As was noted in chapter one, the RAD community suffers from a lack of academic input and hence provide little guidance on producing improved tools

generation languages have been around for a relatively long time. As such, a lot is known about how they work, how best to implement them and the best ways to design them. Of particular interest to this work are the design heuristics used in these languages. By re-interpreting these heuristics for the RAD tool as a whole (i.e. both the graphical and textual aspects of the system), it is possible to define some pointers for the creation of better systems and improve upon the systems profiled in the previous chapters.

Although UIMS specialists apply third generation language knowledge to the language component of their systems, by considering language and interface builders as a whole, important RAD design philosophies can be discerned. This would seem to be an unorthodox approach (based on the literature surveyed as part of this work); indeed the manifestations of problems such as the "dangling else" in HyperTalk would make it appear that much of third generation language research has been ignored completely.

The principles presented below help explain some of the problems with RAD tools discovered in the previous chapter. Each principle will be used, where appropriate, in deciding how to design the next generation of RAD tool.

Universe of discourse

Traditionally the term "Universe of discourse" [Mor93] refers to the different data types a language could process. General purpose languages, therefore, included types permitting the processing of data held by the host operating system and machine hardware. At the time when most third generation languages were designed, this would include strings, characters and numbers.

This principle would seem to account for the observation made in the previous chapter that RAD languages should provide support for multimedia data types. Because the underlying operating system now supports graphics, sound and video, the language should extend its universe of discourse to incorporate these types.

There is, however, another way of using this principle which involves the environment as well as the language component of a RAD system.

It is interesting to make a comparison of the difference between the universe of discourse for the language and the universe of discourse for the graphical environment of a given RAD tool. Certainly for Visual Basic and HyperCard, there is a vast difference. This does not just apply at the multimedia data type level, but also at much higher levels. For example, the interface builder in Visual Basic is able to create and destroy widgets; but this is not possible to achieve using Visual Basic programming constructs. Whilst widget creation and destruction

are possible in HyperTalk, there are widgets used in the HyperCard environment which are not available to the HyperTalk programmer. This results in an internal inconsistency, presenting a tool which appears to provide a certain functionality, yet that functionality is not available from the programming language.

One way the universe of discourse (and indeed other attributes) of third generation languages was tested was to implement the language compiler in the language itself. This became a fairly standard test and languages which did not pass it were, in the words of Levey [Lev84], "beneath contempt".

The equivalent check for a RAD tool would then be to implement its interface builder *using the language component of the tool!* This would not only ensure an appropriate universe of discourse, but make the tool more flexible and freely customisable to those wishing to change it. This would also be a conceptually optimal way to integrate the language and environment.

At the time of writing, there are no RAD tools available which follow this approach[2]. The closest alternative, however, could come from Netscape [Hal97] who have announced the imminent launch of a graphical operating system for Network Computers written entirely in Java — execution speeds aside, this should prove the most flexible mainstream operating system since Unix and C.

The production of this "self-implemented" environment is reminiscent of yet another third generation language technology — *reflection.* The work in reflective programming was undertaken by parts of the functional language [dRCS84] and object-oriented [FJ89] community. The idea behind this paradigm was to provide a language with mechanisms to modify a representation of its state. By viewing the RAD as an integrated whole, self implementation of the environment could give rise to *interactive* reflective facilities — languages which could directly alter their interactive environment. (The term "reflective facilities" comes from Mae's classification [Mae87] which distinguished between fully reflective langauges and those which provide reflective facilities).

Correspondence

The principle of correspondence [Ten77], when applied to programming languages, enables the programmer to treat conceptually semantically identical items in a syntactically identical manner, without having to be directly concerned with small, inessential details. Lack of regard for this principle will produce problems similar to that encountered in the type system

[2]The FaceSpan RAD [Int97] has an interface builder partially implement in the FaceSpan language, but the programmer is not afforded full control.

of Visual Basic, where the conceptually identical text box and variable had different coercion rules applied to them. If there is to be closer integration of language and environment, which would prevent this type of problem from occurring, then better input / output abstractions must be sought.

Reference binding

Reference binding is the one area where the ideas of third generation languages have been applied to RAD systems. In [Thi94] Thimbleby explores the notions of static and dynamic binding in a RAD context. Because RAD tools are interpreted, static binding is unlikely, but they can exhibit a new form of binding, termed *browse* binding. Like dynamic binding (which relies on the state of the run time stack to determine a reference binding), within a visual system, binding can depend on the visual positioning of an object. HyperCard exhibits this behaviour through the use of relative references (`it`, `me` and `this`) so that the value bound will depend on the current position, or visual scoping, of an object (as opposed to its position in the run time stack, as is the case with dynamic binding).

Data type completeness

The principle of data type completeness [Str68] promotes removal of a distinction between "first" and "second" class data types. (The first class types are those defined as part of the language; second class types are those implemented by the programmer.) With the advent of object orientation, this principle has been successfully extended to remove the distinction between first and second class objects. Within RAD tools, which have visual objects, the distinction seems to have re-appeared.

In the case of HyperCard, a distinction is made between the widgets which are supplied with the language (which are available from the tool palette) and widgets added by the programmer as XCMDs (available only as procedure calls). With Visual Basic, programmer VBXs can be added to the environment as "first class" objects, but these visual VBX objects cannot have new instance variables in the same way that the non-visual (Form) objects can. Simply because one object is visible and another is not, there should be no difference in "class."

4.2.2 Guidelines

When compared to third generation languages, there has been surprisingly little written about how to create a novice programming language. The literature which is available falls into two

broad categories: justifications of systems the authors have written ([Pem87], [AS89]) or critiques of these systems ([Coc90], [LH87]).

The first class of literature provides little material which can be used in a general sense; these systems were developed for a specific situation (be it for a specific computer or a specific set of users) and usually as a supported teaching language (not for unsupported novice programmers). The second class of literature is useful only if you have already designed a language which shares faults with a language critiqued in one of these papers! This material is better used in the latter sections of this chapter when considering individual aspects of the RAD tools.

Green [Gre90b], however, has produced *cognitive dimensions* which can be used to measure the effect on learnability of particular design decisions within a programming language. Listed below, these will be used throughout the chapter to aid in design decisions:

1. *Viscosity:* how easy is it to change the system and will those changes produce "knock-on" effects?
2. *Premature Commitment:* forcing the programmer to take a decision before the consequences of that decision can be known.
3. *Role Expressiveness:* how easy is it to determine the semantics of a piece of code?

Besides these measures other, more specific, results from psychological studies of programmers will be used to guide decisions later in the chapter.

4.3 What do programmers really want

If we are to improve RAD tools for the novice user, then it is important, in part, to understand what this class of user actually wants.

4.3.1 Commercial needs

As we have already discussed, commercial software vendors "improve" their software by adding features. They also face a dilemma in producing a completely new novice programming language. If the product is to sell well, then it must bear some similarity to other programming languages or systems currently available — the purchaser is more likely to select a product with which they are familiar. Consequently, vendors of this type of tool have

not been free to develop completely new types of system. Commercially produced systems provide little help in the search for an improved RAD tool.

4.3.2 Programmer needs

One important aspect of the tool is the programmer's *desire* to use it. This applies not just at purchase time, but also how well a tool can support the programmer's desire to explore and learn. Providing enticement at purchase time is obviously a key factor in the design of any system sold at a profit. The less kind observer might suggest that RAD tools are designed to be easy to use, until the point after which the programmer has parted with purchase fee[3].

The importance of a tool's ability to encourage its users is surprisingly high. In a study conducted by Jacob Nielsen [Nie] entitled "What do users really want" Nielsen states that **pleasantness** is one the key attributes users desire from their software — preferable even to working efficiently. This is an interesting result, not only in showing the importance of "pleasantness", but it also supports the tactics of the commercial RAD tool developers, namely to make their products more visually appealing but certainly no more efficient.

This idea of "pleasantness" has been little explored in the context of RAD tools, being mentioned only in a report from a working group on novice programmer systems [MCH92] which states that these systems should provide "instant gratification" and cites examples of directly drawing interfaces and low compilation overheads. "Instant gratification" is really one instance of a more generic attribute, a better term for which might be *Positive Engagement* — a tool can have attributes which are pleasant, but are not instantly so (for instance, the object diagrams of Interviews make specification of inheritance almost fun, but would not constitute "instant gratification"). Also, it is possible for a tool to exhibit unpleasant, or *Negative Engagement* attributes which it would be ridiculous to term "instant non-gratification"!

In chapter one, when explaining the term "RAD tool," it was stated that because novice programmer systems were used for rapid application development, it did not mean that the needs of the novice programmer were identical to those of the application developer. One area where they do overlap is in this attribute of "instant gratification." Because the novice programmer can quickly create what appears to be a fully functioning interface, they are positively engaged by the tool and encouraged to use it further; because the prototyper needs to create a diversity of interfaces to show clients, they will use a tool which lets them rapidly assemble the look of an interface.

---

[3]This is a more cynical view of the first, high reward, section of the plateaued learning curve discussed in chapter three.

Furthermore, several psychological studies [Hoc83, RS85] have shown that static analysis of structure is difficult for novice programmers — they need a language which promotes rapid prototyping (or low viscosity and low premature commitment).

In the following sections, when deciding upon improving particular aspects of RAD tools, these issues of positive and negative engagement will be taken into consideration.

4.4 Deciding on a language heuristic

It almost goes without saying that, when designing a system for novice programmers, the language should be designed to ensure ease of learning. This has been a goal of language designers since the first days of assembler (which was better than machine code, after all). Various paradigms for programming have been developed, all of which have made claims to ease of use. HyperTalk and Visual Basic are both based on an imperative paradigm designed for the novice programmer. Other paradigms for programming language design which one might investigate for improving RAD tools include:

- Functional / Declarative languages
- Visual languages
- Imperative teaching languages
- Programming by example

It could be protested that fourth generation languages should be included in this list. However, programmable fourth generation languages utilise the above language technologies rather than being a separate technology in their own right — SQL is declarative, Excel (since 5.0) has used Visual Basic for Applications (VBA) as its macro language, whilst others [Hil92] are based on a visual language. Ease of programming in this type of system is achieved by providing a restricted, highly domain orientated language.

In addition, interface specification languages such as MIKE [Ols86] will not be directly considered (although individual aspects of these systems will be considered later in the chapter) as they are languages for users who are non-programming interface specialists. The RAD developer, with whom we are concerned and whom we identified in the first chapter, is not an expert in interface design and it is an extra burden to force them to learn both an interface specification language and a behaviour specification language.

4.4.1 Functional

The need for functional languages is best represented in [Bac90]. The claimed ease of use [Cur81] in functional systems lies in allowing programmers to say *what* the result should be, without concerning themselves with the *how* it is to be carried out. Indeed functional programs are often shorter than imperative equivalents. Another benefit of functional languages is "referential transparency[4]", freeing end users from the ravages of side effects; the execution of sub programs cannot be covertly (and confusingly) altered by a hidden global state. Although not yet widely used in this capacity, there have been papers arguing the suitability of functional languages as prototyping tools (e.g. [HJ94] and [Hen86]). This claim, however, is relevant only to the prototyping of information systems by experts; as we shall see, there are some problems in using functional languages for novice programming systems, especially in a graphical environment.

In order to program using a functional language, the novice programmer must learn to control program flow recursively. The work reported in [KA86] revealed that programmers learning recursive flow of control had difficulty and could not transfer this knowledge to imperative programming. Those learning imperative programming could transfer knowledge. Novices can develop mental models of imperative procedures more easily and the researchers postulated that recursion was an 'unnatural' way of thinking. One analysis of functional programming [Mor82] goes even further than this, stating: " Functional languages *are* unnatural to use."

Using Green's metrics, functional languages do score highly in role expressiveness but tend to lag in terms of premature commitment and viscosity. For example, in his paper [Wad92] Wadler, a key proponent of functional languages, describes his disappointment in programming functional languages at being forced to define types before he really knows what he wants and then the struggle he faces in changing his earlier, premature decision. (His solution to this problem, monads [Wad92], will be discussed below.)

Perhaps the greatest impediment to the widespread adoption of functional languages is their failure to cope effectively with input and output — a large failing if the language is being used to create a graphical interface.

Most toolkits are written in an imperative language, making it hard to incorporate them into a functional language. Because the toolkits are external to the language and the widgets of the toolkit have state, it is all but impossible for the language to maintain referential transparency — the user can alter the state of the interface outwith the control of the program and referential transparency cannot be guaranteed.

---

[4]Referential transparency constrains variable scope so that program state cannot be altered by sub-programs which do not explicitly import the variable they wish to alter.

The functional programming community are hard at work to overcome the problems presented by input and output. Text based functional languages, such as Miranda [Tur81], overcome the problem by treating input and output as infinitely long streams of data. By using lazy evaluation, the values in these streams need not be present before the program begins execution.

In [Sin91] and [RS93] Singh uses this abstraction to create two communication channels between a functional language (Miranda, in his case) and an imperative toolkit, to which he has added a functional top layer. This extra layer maps imperative events and calls in the toolbox into calls to functions in the Miranda program. Function calls are passed and results received by the toolkit as text along the communication channels.

A further refinement of this idea was proposed by Carrlson [CH93] and named "Fudgets" (functional widgets). This system provided "Pipe" extensions to Miranda, pipes being information pipes (like those found in Unix) along which information could be sent to different types of widget.

Another approach is that taken by the language Haskell, which uses a form of continuation [All88] called a "Monad." The monad serves essentially as a record containing the state of each component in the I/O system which is passed to every function in the program. Whilst this improves the efficency of the program (referential integrity of the monad is guaranteed, so only ony copy of the monad's contents needs to be held at run time), this is a general purpose abstracion which provides no facilities specific to the abstraction of user interfaces.

The final, and in some ways the most plausible approach, is that embodied in the Clean language [PvE93]. It seems the most plausible because the team involved have produced many real GUI applications, such as word processors and even a version of Tetris (which executes at an acceptable speed, even on a 68k Macintosh). Rather than using closures or monads, Clean allows the declaration of "Unique" variables. These are variables which the compiler guarantees may be altered by only one function at a time. The Clean I/O model has two default "unique" environment objects (one for files, the other for the GUI interface widgets). This allows functional programs to be written (in a functional style) which transform the state of the interface contained within a variable of unique type.

Most literature (for example, [NR94] comparing Haskell and Clean) is far from clear whether there is any benefit to be gained from the application of functional languages to this class of programming task. There are a number of possible reasons for this:

1. Like Macintosh users, there are few moderates in the ranks of functional programmers and they tend to overemphasise the benefits of their systems (a view supported in

[Mor82]).

2. At present, there are no "Visual" functional languages, that is, visual in the way that Visual Basic is visual. Attempts to produce "Visual Clean" [dB93] and "Visual Haskell" [Sin92] have been started, but with no success to date. As a consequence, surveys of UIMS, UIDE and RAD systems have not investigated functional languages.

Because of these reasons, a large portion of the author's time was spent investigating functional languages and their application to GUI programming. The results of this investigation are reported in the next chapter.

4.4.2 Visual

As has already been discussed, the definition of a visual language has been blurred by the recent hijacking of the term by Visual Basic and Visual C++. In this section, "Visual Language" is taken to mean a language with a graphical syntax.

Until recently, visual languages have been too resource intensive to be considered as a serious alternative to text based languages. They are now, however, becoming more widely available on personal computers. Although achieving the same effect as using a text based language, these languages are visually appealing and must surely encourage novice users into programming more than textual languages ever could. They are initially "pleasant."

The most popular application for the visual language is in the creation of multimedia presentations. These systems, of which Icon Author [Aim97] and AuthorWare [Mac97a] are two examples, are iconic. A "program" looks very similar to a flowchart with each type of icon representing a different type of operation. The lines joining these icons represent the flow of control through the program or presentation. Although very good at creating simple systems and more visually appealing to use than a text based language (hence their initial "pleasantness" or positive engagement), the flow chart structure does not scale well.

The first problem is simply one of screen real estate[5]; visual representations are less expressive per pixel of screen occupied (a problem in the development of Visual Haskell [Ree94]). This becomes frustrating as the project expands as it is impossible to gain insight into the overall structure. It also becomes impossible to review programs printed in hard copy when the visual structure is split over more than one page. On screens with high resolution it is often

[5]The effect of this is to produce another plateaued learning curve — as the software developer undertakes the development of more complex systems, they must employ new strategies to overcome the constraints placed on them by the system's limitations.

difficult to distinguish between icons due to small distinguishing details being obscured; i.e. it is impossible for the novice programmer to know that they have used the wrong icon before execution. With text, however, it is immediately apparent if you cannot read a word correctly. This is a viewpoint supported by Mayer [May81] and Green [Gre90b] who discovered that programmers can understand textual syntax more quickly and accurately than any current form of graphical syntax.

The metaphor of the flow chart also seems curious when considering the types of task to which these systems are applied. Flow charts were conceived in the days of third generation languages whose programs had a single flow of control with users following a prescribed route through the program. It seems strange, therefore, that the flowchart should be used as a model for the control flow for a program which is likely to undertake parallel tasks (for example, animate a picture and play a sound) and support hypertext forms of interaction. Even when flowcharts were adequate to describe a program's behaviour, there was some evidence [She81] which showed that flowcharts provided no detectable benefit to the programmer.

By using this flowchart metaphor, these systems force the programmer to think explicitly about the facet of programming they are likely to find most complicated. This is also a form of premature commitment, as it forces the structure of the program to be decided before screen design can begin.

In considering measures, such as premature commitment and role expressiveness, it must be remembered that these systems are semantically equivalent to general purpose programming languages, only the syntax is different [Ree94]. It is therefore inevitable that the language will reflect the strengths and weaknesses of the paradigm on which it is based.

The frustrations with functional languages expressed above are not mere conjecture. During the work conducted for this thesis, the author joined a team using visual languages to create various CAL packages for a TLTP project [Mat97]. This was deemed necessary to understand the suitability of visual languages for implementing non-trivial programs. The issues raised above were expressed not just by the author, but also by the other members of the team as an initial fascination with visual languages eventually turned to frustration. The results of these experiences are published as [MC97].

Of course, there are other forms of visual language, but there is no evidence that any are better than their textual counterparts. Claims are made in Shu [Shu88] to the superior usability, but there is little evidence (either in the book or in independent literature) to support this view. If visual languages are to succeed then there must be a programming paradigm developed to specifically exploit the attributes of a visual syntax.

One area where visual languages do have an opportunity to improve over textual languages in

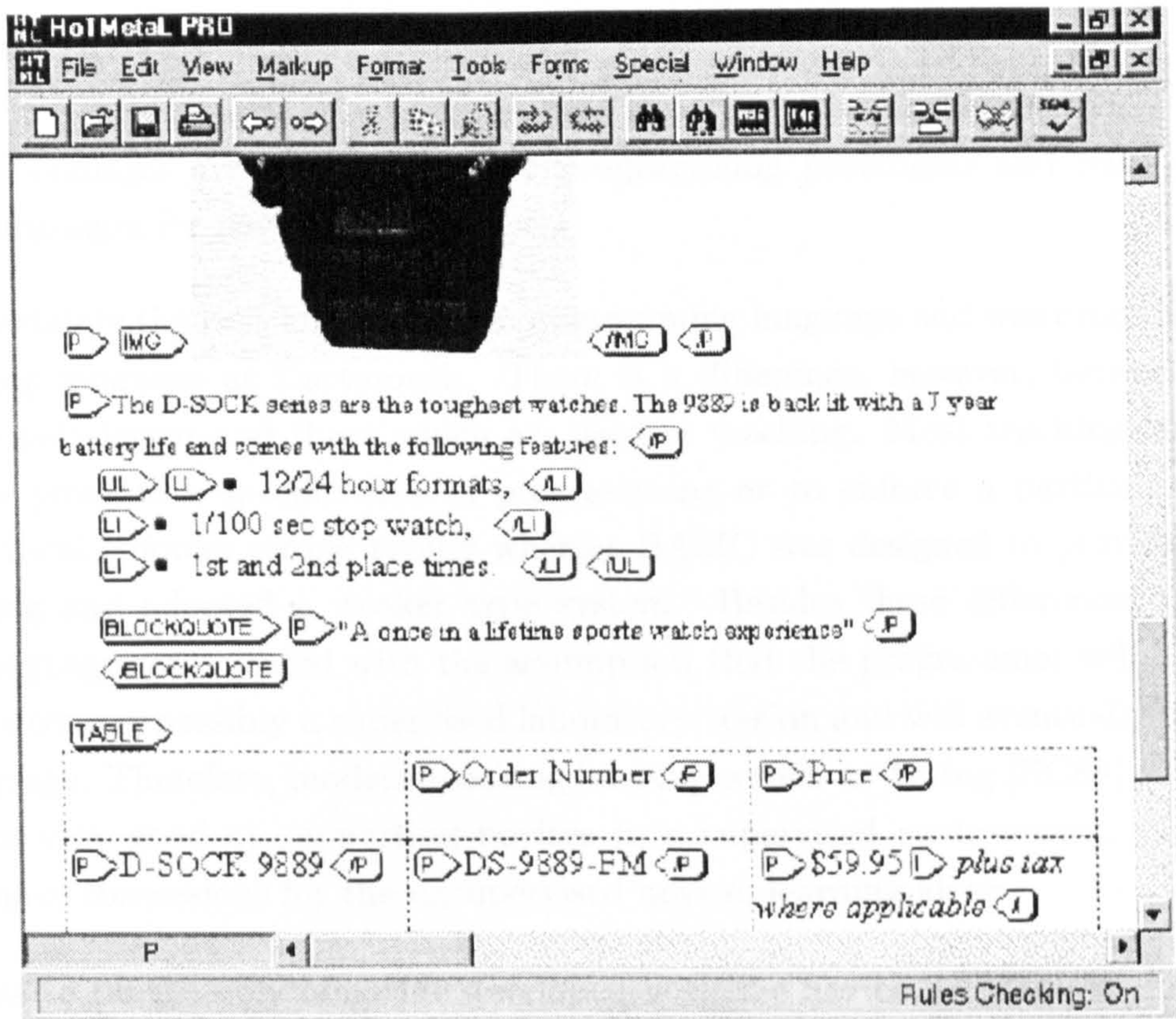


Figure 4.1: Within the HoTMetaL environment, graphics and text can be freely edited.

the domain of GUI programming, is by exploiting the graphical nature of their editor. Because the graphical aspects cannot be viewed "in-line," textual programs which are responsible for displaying and manipulating graphics require the programmer to mentally visualise the effect of the program. This requires complex mental skills, as anyone who has tried to achieve a particular layout using HTML can attest to. Until recently, this form of hybrid system has not existed (i.e. one which displays graphics and text simultaneously). The historic precedent of visual languages being separate from textual languages should be overcome — as systems like HotMetal [Cor96] have shown they need not be mutually exclusive. See Figure 4.1. In fact, one area where visual languages, such as Icon Author and AuthorWare might be easily improved is in the replacement of the flow chart with some visualisation of an event based model.

As a final note, the adoption of purely visual systems seems historically flawed. If a lesson can be learnt from the evolution of natural language representations, then it is perhaps that *purely* pictographic systems should be discarded in favour of textual systems. One is left feeling that the development of iconic systems is pursued simply because the technology now exists to do so. More cynically, one might suggest that it is easier to market a visually engaging product. The ideal system would be one which can exploit the terseness of text with the spacial and representational strength of visual systems.

4.4.3 Imperative teaching languages

Imperative languages are the oldest of all programming paradigms and have produced a variety of languages for novice programmers.

BASIC is certainly the best known novice programming language and was originally designed as a teaching language at Dartmouth. There is a difference, however, between languages which are easily learnt and those which are used in teaching. Most teaching languages are designed to promote a specific area of programming or to enforce a particular style. For example, Pascal enforces strong typing whereas BASIC was designed to provide interactive programming and adopted a weaker type system. Besides these differences in emphasis, teaching languages are created with the assumption that the programmer will be supported by a lecture course, possibly a supervised laboratory session and will eventually program in a "real" language. Therefore, modern teaching languages such as Turing [HC88] and S-ALGOL [Mor93] are very good at supporting novices in a supervised environment. They do not, however, make concessions for the unsupervised novice, learning alone.

ABC claims to be the only language developed with the novice programmer in mind. It is persistent and comes with a "language aware" editor which automatically completes brackets, quotations and nesting statements. It is also a "complete" language, with data structures and user defined types which together provide support for most programming tasks. These facilities indeed make it ideal as a first language, but it currently does not support any form of graphical user interface. Moves have been made by the ABC team to provide graphics primitives [ZZ92], but no attempt has been made to use it as the basis for a RAD tool. From ABC, however, the importance of integration in enhancing ease of use is easily seen — integrating the editor and language allows command completion and bracket balancing, whilst integration to the physical file store provides persistent data types.

Currently, the most applicable imperative paradigm to interface programming would seem to be object-orientation. As direct manipulation interfaces use widgets which resemble physical objects (buttons, gauges, dials etc.), it is logical to consider the role of individual objects, rather than deconstructing the interface on a functional, or any other basis. Combining objects with event based programming frees the programmer from worrying about the overall program flow of control[6] — a key problem in novices' comprehension of programs [NM90].

By using an event based language, programmers are again free to think about the behaviour of individual objects in the interface, again a view supported in psychological studies of programmers [Gre85]. Event programming also aids incremental development, as the behaviour

[6]Of course, just because object orientation does not force novices to worry about flow of control, it does not guarantee that they will create a sensible program!

of objects can be specified for individual events, rather than needing a definition for responses to every possible event. The suitability of event based object-orientation is evident in its overwhelming pervasion of GUI programming tools.

It seems clear, therefore, that, for the time being, this is the most appropriate paradigm to adopt in the creation of a RAD tool.

To finish this section, we shall use the words of the ACE team [Joh93] in their discussion on designing their end user system:

> Finally, it is not enough to have a good language. There must be a supportive development environment...The bottom line is that end-user programming is not just a language issue. It requires careful integration of a language with a visual framework.

4.5 Bridging the gaps

In the previous two chapters current trends in RAD tools have been examined, discovering that they owe a lot to their procedural, third generation ancestors. We have also seen how the current lack of integration (both conceptually and at a lower level) between the various elements of language, toolkit and environment, causes problems for prospective programmers. To improve the situation, therefore, requires an entirely new tool which better integrates these elements into a higher level system, designed for this particular form of programming. Ideally, the measures of integration identified in chapter two should be as high as possible — i.e. there should only be one entity, not a separate toolkit, language and creation environment. This move towards greater integration is really just a continuation of the historical trend, starting with UIMS, then continuing with UIDS and, most recently, RADs.

In [Nie93] Nielsen states that end user programming has declined over the last 30 years as the gap between text languages and visual environments has widened. He suggests improvement through a script system built into the operating system, affording the user full control over their environment. Certainly, this will make creating applications easier once the language is learnt, but it does little to move the language and GUI abstractions closer together. In fact, AppleScript provides much of what Nielsen desires, but does not provide any higher level abstractions than can be found in current systems. What is important, therefore, is to examine how language, toolkit and environment can be more closely integrated. This examination is presented in the following sections.

4.6 Bringing the toolkit and environment together

Merging the toolkit and environment is the most straightforward integration to be considered, largely because these elements are already strongly linked. As we have seen with HyperCard and Visual Basic, these systems often evolve as an extra layer on top of the toolkit. This integration is especially enhanced by the creation of object-oriented toolkits and the creation of component standards (such as OCX and VBX).

One of the great strengths of Visual Basic over HyperCard is the way in which toolkit widgets are permanently displayed in a palette which exploits the object nature of the toolkit and is easily extended to include new controls. By adhering to the principle of data type completeness, the VBXs added by the programmer have the same "rights" as those supplied originally. Extendibility is important not only in terms of accessibility to the programmer, but improving the tool's use in prototyping — if a particular widget is unavailable, then that widget cannot be used in any prototypes created by the tool.

Furthermore, by providing a property browser, Visual Basic reduces cognitive load on the novice programmer by removing the need to learn the properties of particular widgets. This will also inevitably promote exploratory learning. What would, of course, be welcome is an equivalent method browser for objects. In addition, these browsers should present every method and property of the widget — some systems (Visual C++ being the most guilty) have hidden properties, whilst others have properties which are only available at run time or only available at design time. This sentiment is echoed by the DRUID [SKN90] development team who discovered that:

> Many interface builders are only add-ons to certain toolkits and, ever worse, only provide parts of their full functionality.

Needless to say, this is undesirable and results in an inconsistent universe of discourse.

4.7 Bringing the language and environment together

The solution to the problem of merging language and environment has already been discussed when considering the application of a consistent universe of discourse — i.e. the language component of the system should be used to implement the environment. Currently there are no RAD systems written in this way, although some Macintosh RAD tools, such as FaceSpan [Int97], which are based on AppleScript, permit control of the visual environment. This is

more by accident than design as the visual environment, being a Macintosh application, must respond to AppleScript events (or else the operating system could not communicate with it).

This self-implementation gives the programmer full control over their environment. They will no longer be mis-led about the capabilities of the system, as everything that appears in the environment must be reproducible by the programmer. If created properly, portions of the interface could be re-used as objects in any new applications.

Whilst it would have been interesting to have produced such a system, it was felt that the scale of such a task is beyond the confines of this thesis. It is hoped, however, that the worth of this idea is self-evident.

What remains to be resolved, however, is the integration of language and toolkit.

4.8 Integrating language and toolkit

Whether phrased as "lack of appropriate I/O mechanisms" [Mye92b] or expressed as a desire for "a computer language that embodies UI abstractions" [Hay84], current RAD languages do not provide high level constructs for toolkit control. Given that the principle of abstraction should be adhered to within RAD languages, what are the appropriate abstractions?

In third generation procedural languages, the abstraction for input and output was the *streams* concept. A program could expect to receive characters from an *input* stream and place characters on an *output* stream. Using this abstraction, the programmer did not need to worry about low level details such as file or keyboard buffers.

Can the streams concepts still be applied to the RAD language?

Input

The abstraction used for input to a RAD language is a stream, but this time it is an *event stream.* Input devices (typically a mouse or keyboard) can generate events which are placed on an event queue and dealt with in a FIFO fashion. Mouse events are dispatched to the widget directly underneath the pointer and keyboard events are dispatched to the widget "in focus"[7] (see Figure 4.2). This is very similar to the character stream of the traditional third generation language and would seem to provide an appropriate abstraction.

[7]There are exceptions, such as meta-key events which, for example, open a new document and are sent to the menu handler, rather than the focused widget.

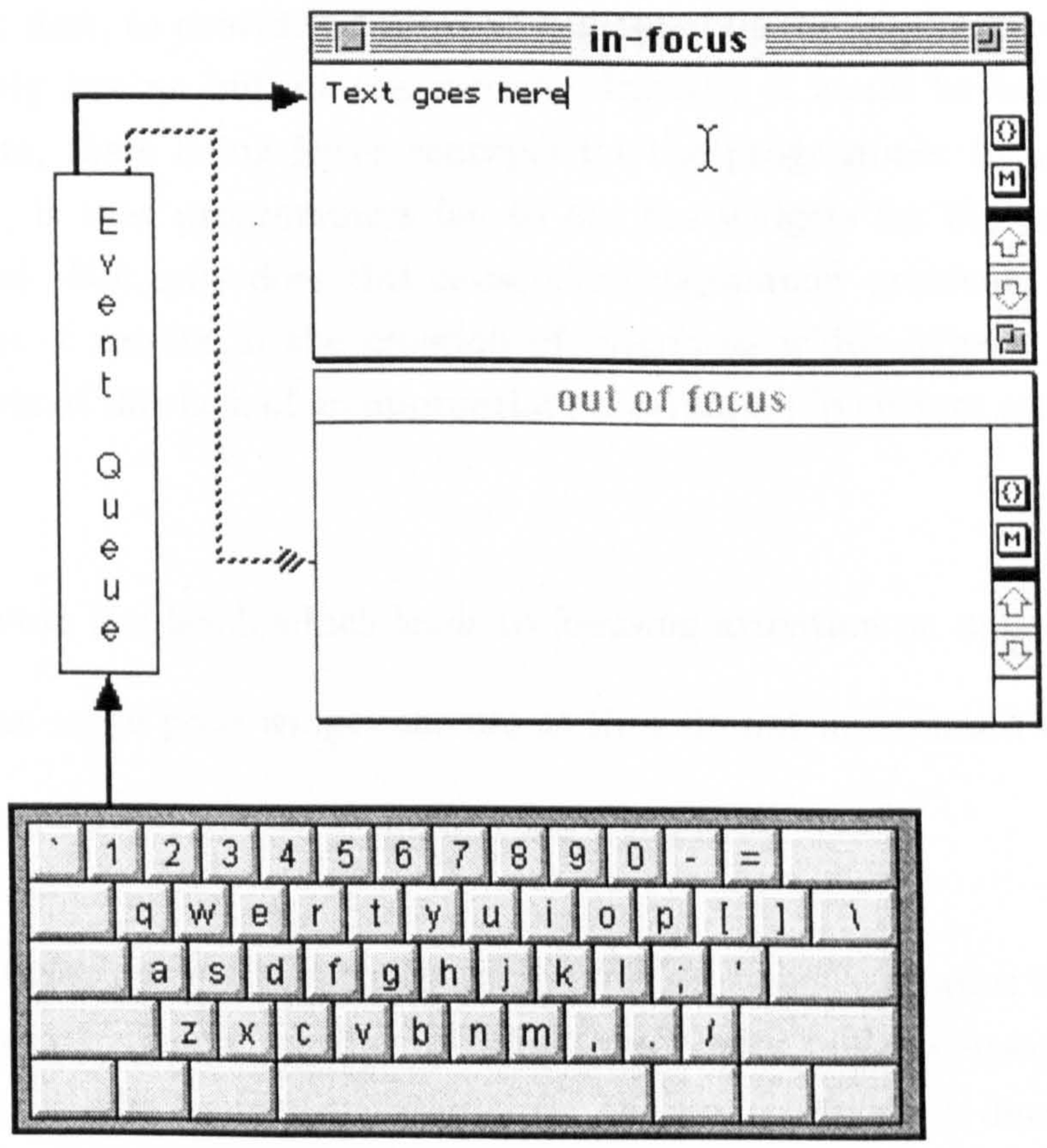


Figure 4.2: The key presses from the keyboard generate events which are placed in the event queue. These events are then dealt with the "in focus" widget (in this case denoted by the window with the horizontal lines in the title bar).

Output

The "output" or, perhaps more accurately, the evidence for the existence of a RAD program, is the graphical user interface. This is comprised of toolkit widgets which are controlled from the program's code. These widgets are used for a diversity of purposes: displaying graphs, receiving text strings, invoking procedures etc. Despite the difference in purpose, the abstraction used by the language to control them is identical across all widgets. Effectively each widget is treated as no more than a general purpose object with methods and properties.

This may seem, at first, to provide an ideal abstraction but the implication for the language is that it effectively has *no* output abstraction. Initially it could be believed that this is a positive attribute, there being fewer concepts for the programmer to understand. What happens, however, is that programmers fail to use the widgets for the purposes for which they were designed. Not only does this cause the programmer problems in deciding which widget to use, but it results in the creation of interfaces which behave in an unexpected manner. The effects of this lack of an appropriate abstraction in current systems are reported in [Joh93]:

- Toolkits are too low level, which leads to focusing attention on appearance issues.
- Designers can make poor widget choices as they do not understand what they are for.
- Widgets do not mesh well with application semantics.

Books written to teach novices how to use these tools do little to improve the situation. Most of these books teach by using tutorials in which the reader builds consecutively more complex applications. Whilst more motivational for the programmer, it does little to promote sound programming practice or good interface design. In fact, it is only in the Human Interface Guidelines, sold to professional developers, that the actual function of each widget is discussed. Even then, whilst some guidelines [Mic90] provide a comprehensive and straightforward explanation of widget usage, others provide only vague and sketchy explanations [Com93].

Again, the problems of using this third generation technology in an inappropriate manner are manifest; obviously, another abstraction must be found.

4.8.1 Constraint based programming

One way to integrate toolkits and languages is through the use of constraint programming [Sut63]. Constraints permit the programmer to define relationships between objects. For example, this would allow a programmer to define that an arrow should always point to the left edge of a certain box; when that box is dragged by the user, the arrow will automatically move to the box's new position. This is essentially the same approach as spreadsheet programming where the behaviour (value) of cells can be specified as an interaction (calculation) on some other cell.

Systems such as Borning's Thinglab [Bor81], implemented in SmallTalk, support simulations of physical systems, where the interaction between objects is specified in terms of physical constraints. Another domain specific use of constraint programming can be found in graphics packages [OA90], where the designer can set constraints between graphical objects (e.g. **this** line should always be at seventy degrees to **that** line.)

For interface creation, one example of constraint based programming is the use of pre and post conditions, as reported by Foley [GF92]. Within this system, each interface object method is assigned pre and post conditions, stating when they should be invoked and what must take place before they can terminate. Within the Garnet [Mye92a] UIMS, Myers has built a more general constraint solving system, based around the spreadsheet programming model which he claims has greatly aided interface programming.

There is no doubt that the setting of constraints between objects can be a natural way to conceptualise certain problems, but it still does not provide a general abstraction for all interface programming problems. Constraint programming also has some shortcomings which must be overcome before they can be used in a general programming tools.

For instance, the Foley system requires that the full functionality of the interface is understood in terms of a finite state machine (FSM). Besides the problems in scalability of FSMs, the building of an FSM is a premature commitment and would act as restraint to rapid prototyping if adopted into RAD tools.

Myers admits to problems within the Garnet constraint solver, especially in the area of debugging constraints that have gone awry and produce unexpected behaviours. Constraint solvers of this nature also tend to be large and complex, greatly increasing the hardware requirements of the host machine. At present, they do not seem to be sufficiently mature to be incorporated into a RAD system, although there is much research being conducted on the design of new constraint engines [VZ92] and the integration of constraint languages into interface programming systems [FBB92].

Interface Specification Language

One approach used in professional development environments is the provision of a separate interface specification language. This language is used to specify, at a very high level, the types of interaction encountered at the interface. Typically, this specification is conducted by an interface design specialist. The specification is then compiled into a more traditional programming language to allow the functionality to be added by an application programmer. Typical examples of these systems include Olsen's MIKE [Ols86] and the Lean Cuisine system of Apperley [AS89].

The benefits of adopting this approach include:

- The interface designer does not need to learn a programming language, only the interface specification language.
- Because the interaction is specified at a high level, the specification compiler can automatically choose the most appropriate widgets.
- The compiler can automatically apply other design guidelines such as layout, corporate styles etc.

These items are obviously desirable in any interface creation system and would address the concerns of Johnson stated at the start of this chapter. Can an interface specification language, however, provide a suitable abstraction for a novice programming tool?

However, adopting this approach would require a novice programmer to learn both the interface specification language **and** a behaviour specification (programming) language. The inclusion of another language in the specification of the interface might increase viscosity as the new language could complicate the compilation cycle to:

> specify interface → compile interface specification into programming language → modify language → compile programming language.

Not only are there time penalties introduced in processing time, but as with all back end compilation systems, the programmer is forced to build their code into a structure created by the specification language compiler which is not necessarily how the programmer would like their code to be structured.

The ideal abstraction, then, would include those attributes listed above, but somehow incorporate the specification of interface widget interaction into the programming language.

4.8.2 Specifying interface semantics within the language

Having examined the possible ways in which an abstraction of the toolkit could be achieved, there would seem to be no optimal solution. In light of this, several attempts were made to design and incorporate high level user interface abstractions into an object-oriented, event based language.

The button

Once chemistry students understand the atom, they can be taught about the periodic table, atomic numbers, how atoms form elements and how elements combine to form compounds. Perhaps in providing a model for how widgets interact with each other and the outside world, it may be possible to discover some fundamental, irreducible widget (of which other widgets are more complex forms) and examine how a compound of widgets combine to form an interface. This "reductionist" approach would seem to be supported by Atkinson [ABC+84] who observes:

> One important iteration is to improve the parsimony of concepts...the designer must investigate whether there is a more primitive concept that will serve both roles.

This simplification would then give the novice programmer a building block which could be learnt quickly in the beginning, with more complex combinations of the element being learnt as experience and needs increase. What, then, is this fundamental particle?

HyperCard comes closest to adopting this approach, having basic "field" and "button" objects which can be specialised into scrolling fields, radio buttons, check boxes and pop-up menus. In addition, HyperCard also has "background," "card," "stack," "menu" and "HyperCard" objects. This list can easily be reduced to "field," "button," "background" and "menu" on the grounds that "HyperCard" is conceptually a collection of "stacks" which is a collection of "backgrounds" and a "card" is a particular instantiation of a "background." Is it possible, then, to reduce this list of four even further?

The most readily reducible object is the menu, which could, arguably, be reduced to a list, or array, of buttons — a list of command buttons could be considered equivalent to a menu invoking commands (a typical "File" menu) and a list of radio buttons or check boxes equivalent to a menu for setting options (such as the "Style" menu in a word processor).

Another reducible object is the field, which could be reduced to a form of button, which is capable of holding text. Certainly both fields and buttons are identical in that they can contain methods which respond to events except that fields also have extra, text related properties.

This now reduces the list to the button and the background. Interestingly, a background could also be defined in terms of a button — a button on which other widgets might be placed! We could therefore conclude that, if it is sensible to have a fundamental widget particle, then that particle is the button.

Whilst this looks like an appealing theory initially, it rapidly becomes inconsistent and unworkable. To examine the "solution" more fully, a sample implementation of this abstraction is attempted in the next chapter.

Equating language concepts to interface concepts

One way of integrating toolkit abstractions into a programming language would be to determine if there was an overlap between the semantics of the programming language and the semantics of the interface components. From the analysis of the Visual Basic type system, realising that the text box widget and string variables were semantically equivalent, it would seem that this approach may prove fruitful.

There are hints of this approach throughout research into programming systems for experienced programmers. In systems such as PICASSO [RKS+91] and D2M2edit [dBFM92] programming language constructs (with which the experts are familiar) are used as metaphors to explain the semantics of the interface (with which the programmer may be less familiar). If there is a conceptual overlap, it should be possible to build a language with abstractions which can be used to represent both application and interface semantics.

Whilst the semantics of most programming languages are well understood, the semantics of interface widgets are not. Fortunately some recent work carried out at Hewlett Packard [JNZM92] as part of the ACE project has gone some way to addressing this problem. After surveying a wide variety of interface widgets, they concluded that a widget's semantics of selection widgets could be classified as belonging to one of the following four categories[8]:

- Select one from a small set of discrete values
- Select several from discrete set of values

[8]The semantics of "containment" widgets, such as labels and text boxes will be discussed later.

- Select a single value from continuous range
- Select a subrange from large range

This work was again in the field of expert programming systems, the intention being to create new classes for C++ to improve its ability as an interface prototyping language. Within these new classes, the different types were termed "selectors" to which one could add a "presenter", the presenter being responsible for how that particular selector appears on the screen. By creating the selector / presenter split, the ACE system gains a number of benefits:

- Allowing the programmer to concentrate on the behaviour of the interface, without becoming lost in appearance details.
- Allowing multiple presentations of the same data.
- Making provision for the automatic selection of widgets.
- Increasing the "role expressiveness" of the language.

Using these classifications as a basis, the creation of a new language was attempted; this language uses abstractions which can be "presented" by the system to create interface widgets.

4.9 Individual Components

Having decided upon a language heuristic and how to integrate that language with a toolkit, it now remains to address lower level concerns and present ways in which some of the problems of RAD tools identified in chapter three might be overcome.

4.9.1 Language Structure

In order to create a consistent object model, the inconsistencies of placing a procedure based language (either Visual Basic or HyperTalk) in an object environment can be overcome through the use of a pure object-oriented language. Indeed SmallTalk [Gro81] and, more recently, JavaScript [Fla96] are pure object-oriented languages advocated for end user programming.

The use of such a language will build on the benefits of object-orientation found in current RAD systems but will also remove the conceptual baggage inherent in current systems. An

example of this baggage is provided in the following box. Other, more detailed aspects of this language are considered in subsequent sections.

For example:
HyperTalk maintains the distinction between function and procedure declarations. All these different constructs could be reduced to an event handler, where an old style function or procedure call is just an event handled by the appropriate handler. (Even in the previous decade, the distinction between function and procedure was being called into question [Hab73].)

4.9.2 Syntax

It has been argued that the choice of syntax is crucial to the programmer's understanding of a language [LWSS80]. In this paper, Ledgard asserts that the syntax of a language is as important as its semantics — based on the discovery that even "expert" programmers could not distinguish between language syntax and language semantics.

Having decided previously that HyperTalk's syntax is overly cumbersome, what guidelines exist for the creation of a new syntax for novice programming languages?

There are some convincing results in the analysis of language syntax, such as Shneiderman's demonstration that GOTOs are harder to understand than nested control structures [Gre85]. This provides psychological support to Dijkstra's arguments that the use of GOTOs lead to poor software quality. The result was also successfully replicated by Green [Gre80] who, in turn, discovered that nested control structures were more easily understood than several suggested alternatives. Fortunately, the removal of GOTOs and use of nested decision structures are pieces of research which were implicitly taken into consideration when Visual Basic and HyperCard were being designed and which should be carried through to any new novice programming system.

Other suggestions made by Green relate to the role expressiveness of the syntax. Suggestions are made for the replacement of the "=" symbol for assignment; perhaps by "←". Certainly the use of "=" and "==" in the C family of languages would seem to invite ambiguity[9]. Another legendary candidate for abolition from a new syntax is the use of the ";" (or any other symbol, other than "newline") for statement conjunction or termination. Again, fortunately, neither Visual Basic nor HyperTalk use any explicit line termination or block conjunction symbols. Although modern compilers can warn and spot many of the mistakes caused by

[9]With the adoption of this convention into Java, it seems that this is a design decision which will be with us for a long time.

confusing syntax, this is no substitute for a syntax which is well designed to begin with.

There comes a point, however, where it is impossible to provide prescriptive guidelines in the use of specific symbols. In studies conducted to determine an "ideal" syntax for a command language, the results showed that the probability that two people would choose the same name for a command was around 20% [FLGD87]. It would seem impossible, therefore, to provide an ideal syntax; the best that can be achieved would be to build a synonym facility into the language.

One principle which has been suggested for use in designing a language syntax is the principle of *least surprise*, Soloway and Rist [SE83] (also called the principle of *least astonishment* [Thi90]). Essentially, this principle can be stated as follows: *language structures which are used for similar tasks should have similar structure.* Hardly revolutionary, but this principle gives an insight into another source for syntax design principles.

One book constantly referenced in the creation of this text is [SW79], a guide for the use of grammar rules and the elements of a successful writing style. This book also contains the principle of least surprise, stated as:

> **Express co-ordinate ideas in a similar form.** This principle, that of parallel construction, requires that expressions similar in content and function be outwardly similar.

As an example of this principle, the Beatitudes are quoted in [SW79] as:

> Blessed are the poor in spirit: for theirs is the kingdom of heaven.
> Blessed are they that mourn: for they shall be comforted.
> Blessed are the meek: for they shall inherit the earth.

It would be an interesting study to examine similar guides to see if any other grammar rules might be translated to syntax design.

Finally, it is worth considering the effect of the syntax in terms of user engagement. The effect of a visual syntax has already been discussed and, for the time being, does not seem to be an appropriate solution. HyperTalk engages the user by presenting familiar words to the programmer but suffers from being poorly specified and overly verbose. One way in which these problems might be overcome and the positive engagement of English preserved is through the use of a properly specified grammar (for example, LALR) and a command completion system, similar to that used in ABC. Such a syntax would provide the initial "English" familiarity of HyperTalk, and perhaps remove the later frustrations.

4.9.3 Data types

In designing a type system for this type of language, there is a fundamental conflict in providing a type system which is not prohibitive to the novice, but which supports the development of "correct" programs. Visual Basic attempts to solve this problem by effectively providing two different type systems: variables can either be declared to be of a particular type (whereupon strict type checking is employed) or of a variant type (whereupon variables are effectively typeless and coerced automatically). Although this is one solution, it requires that variables be redeclared should strict typing be required — a high viscosity solution.

Other systems, such as ACE [Joh93], allow the programmer to treat type mismatch alternately as:

- an error, whereupon the program is halted
- as something to be ignored
- causing the invocation of automatic coercion

It is not described how this is implemented, but from the manner in which it is reported, it may be safely inferred that this is an attribute, or preference, of the compiler. Although less viscous, the type checking is not fully under the programmer's control — they are only afforded the options presented by the compiler. In the next chapter a prototype type scheme will be considered which allows the programmer to implement their own type scheme.

4.9.4 Data structures

The data structures should reflect the type of task the user wishes to undertake. Although more efficient in terms of storage, the static array forces the user into premature commitment. Dynamic arrays are obviously preferable.

One of the problems in providing dynamic data structures in third generation languages is that they require the use of record and pointers — concepts which the HyperCard team, at least, decided were too complex for the novice programmer. Functional languages, however, provide "list" data structures which require no explicit memory control (through pointers or any other device). Lists would seem to be an ideal way to store data in a sequential, dynamic structure. Although operators on lists in functional languages require recursive programming, these could be replaced with imperative equivalents. One language already adopting the list is AppleScript, through use of a `FOR ..IN` loop which can process every element in a list (really the equivalent of a map function).

As stated in the previous chapter, the language should also include structures and operators to cope with multimedia data types, such as images and audio samples. This could also prove a significant factor in improving the execution speeds of applications written in the language as microprocessors are now being provided with machine code optimised for exactly this type of data. The Intel Pentium MMX [PWW97] already provides SIMD instructions to modify bitmaps and audio samples; also included are optimised "saturation arithmetic" operators for rapidly altering bitmaps.

Finally, making data structures persistent frees the programmer from the problems of mapping a program data type onto the file types provided by the operating system. Instead, each application maintains a permanent object store, akin to a multimedia database [Bla97], from which values may be retrieved.

4.9.5 Error reporting

Reading compiler errors is a thoroughly daunting task for the novice programmer. Many systems will happily produce negatively engaging reports, such as "Compilation aborted - 96 syntax errors reported." Novice programmers may, understandably, infer that their program contains 96 mistakes which they need to correct. Furthermore, this distress can also turn to bewilderment as 95 of the reported errors "disappear" when one semi-colon is added. To experienced programmers (i.e. those who are implementing the error reporting system), long lists of errors are not intimidating and can be a source of amusing anecdotes ("you think that is bad, I once compiled a program in VAX COBOL which..."). Lest it be felt that such errors are an anachronism, the author has witnessed the latest version of an award winning Macintosh (renowned as the friendliest computer available) Java (latest generation, object-oriented language) compiler which reported approximately 1000 syntax errors in one compilation cycle! A compiler designed for the novice programmer should, at least, provide a facility for limiting the number of errors reported.

The actual form of the error messages produced (regardless of their number) can be confusing and even offensive (*abort* has some obvious negative connotations [dB84]). Most compiler errors also tend to offer information which is of more use to those who implemented the compiler than those trying to use it; "Stack overflow", "Attempted divide by zero" being two common examples. The novice programmer is forced to cross reference these bizarre messages with the manual in order to decode them.

Most modern development environments provide on line documentation which can be automatically indexed from the dialog reporting compilation errors. This goes some way to improve the situation for novice programmers, but the help provided is descriptive, rather

than prescriptive. Some systems, however, provide tutorial type documentation (CodeWarrior [Met97] for example provides electronic books, such as "Learn Java on the Macintosh" aimed at novice programmers) which could easily be cross referenced in the same way as the code library references books. Novices could then explore sample code, looking up the concepts they do not understand, or have the compiler reference the appropriate section when an unexpected compilation error occurs.

4.9.6 Removing use and build modes

Unlike so many aspects of user interfaces, modes are a clearly identifiable source of misconception and error. Mode errors arise because the user is unsure of the current state of the application [Mon86] due to insufficient feedback. Even if these modes are apparent to the user, they may appear confusing and at odds with user understanding of the functioning of an application. Sadly the environments of RAD tools suffer from every form of mode error.

One way of tackling the mode problem is not to tackle it at all! After all, every programming language has a design and execute phase. Visual Basic, therefore, has very clearly defined design and execute modes. HyperCard exhibits application wide design and execute modes, but provides a more gradual introduction to GUI programming. Within HyperCard, programmers choose from a selection of tools which allow them to alter the mode of interface interaction. Clicking a button with the browse tool will cause the button to respond with normal, execute behaviour. Clicking the button with the button tool, however, will allow the appearance and behaviour of the button to be altered. They can migrate smoothly from an interface user, through various levels of tool complexity (five in all), until they can confidently program the interface.

The main problem with both these solutions, however, is that no attempt has been made to remove the mode from the interaction: novice programmers will still see inconsistent widget behaviour. This problem is exacerbated by systems which fail to adequately convey which tool, or mode, is currently active. Studies with interface builders report that when using some systems, even expert programmers can have difficulty determining which mode is active [HS95].

An alternative to having a system wide mode is to introduce modality on a per-object basis. This would seem to be more in keeping with the real world, where it is perfectly feasible to paint (design) a wall and listen (execute) to a radio. It also has the benefit of removing the tedious mode changes encountered when developing an interface in a system which uses a global mode change.

An alternative to mode based interaction is the provision of affordances on objects. This has been used successfully in applications such as window managers, where there is a tightly constrained set of customisations a user can make. By adding close, zoom and resize boxes to a window, the user can perform modeless customisations on that window. The affordance solution, however, cannot be easily applied to a programming environment. Adding customising affordances to each widget would clutter the interface and hopelessly confuse any end user. (There is also a problem in customising the affordances — for example, how do you change the size of a close box? Provide a meta-affordance?)

In [SS92] Smith proposes a solution based on buttons which apply customisations to widgets. An interface programmer / user can drop customising buttons on objects, which change that object's behaviour. The customising buttons, argues Smith, can be thought of as tools, and thus the solution is very like the user's real world experience. Certainly, such a solution does not introduce any mode, as the "tools" are part of the executing interface. By providing tools of varying complexity, users can easily make the transition from simple to complex interface alterations. The idea of a button representing a tool, however, does seem to unduly stretch the programmer's experience of the real world. This does not bother Smith, who seems to equate an intuitive solution to one in which objects have no modes.

In the next chapter a new metaphor is presented which presents design and use modes which really are consistent with the user's experience of the real world.

4.9.7 End user customisation

Because HyperCard does not enforce the distinction between design and execute modes, there is no meaningful distinction between programmer and user — (notwithstanding security) the end user has access to the same tools as the original programmer and can endlessly customise the application. If the original programmer wishes, they can control the level of customisation by the use of the "userLevel" property.

This customisation process is further supported by the direct relationship between interface and application code structure. Much research in UIMS was aimed at separating the user interface semantics from the underlying application [Sze88]. In certain environments, especially in large projects, this is a desirable property. The inclusion of another layer allows the development work of the application functionality to be distinctly separated from the development of the user interface functionality allowing different programmers to work on each task. Unfortunately, this extra layer between interface and application distances the end user from the application code making it harder to alter the underlying functionality.

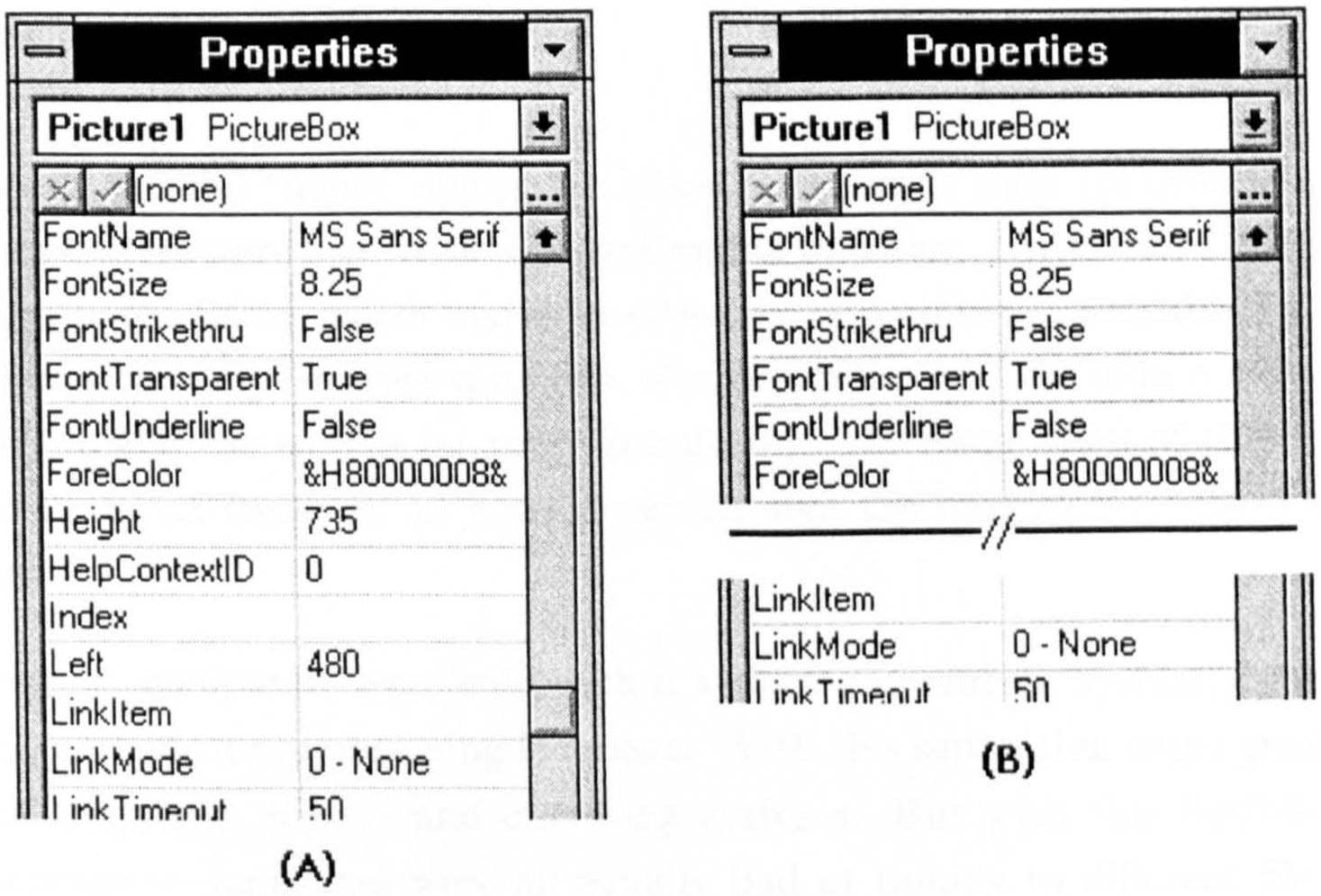


Figure 4.3: In **(A)**, all the properties are available to the user. In **(B)**, however, the user has less rights and is restricted in what they might customise.

Assuming that the end user wishes to customise the interface [Mac91], what other features should a RAD tool provide to aid this process?

Even with a "user level" mechanism, because of the low level abstraction currently used for interacting with the toolkit, the attributes of widgets are either all available or all unavailable to the programmer. To produce more user customisable software, it should be possible to assign priorities to attributes of widgets, and the application as a whole. The priorities should be dynamic (i.e. determined by the programmer, not by those who created the toolkit) thus enabling priorities to be altered depending on the customisation needs required. This can then be reflected in the property browser as in Figure 4.3.

Furthermore, although object orientation ensures that the event handlers and attributes for a particular widget are encapsulated with that widget, the same cannot be said for the overall application structure. This means that, were a customisation tool to be provided, the user would intuitively know where to look to access the attributes of a widget, but altering more general attributes would be less intuitive. If the structure of the interface, however, were to more directly reflect the code structure, then customisation would be more straightforward. Using the user interface abstraction which relies on overlaps in interface and language semantics, it is possible (as will be seen in the next chapter) to write programs where the application code structure mirrors that of the interface.

4.9.8 Editor

In the beginning was the "home" computer, based on a simple eight (or even four) bit processor. Home computers were sold with a programming language interpreter (typically BASIC) incorporated into the ROM, which was quite often the only software available for the machine. Because this was the only software available, the interpreter had to contain disk management and editor commands *as well as* language primitives. The integration of these various components was never an issue, as all access was through the language — there was only one environment.

Later, "personal" computers were sold with a separate operating system, basic application packages and perhaps a programming language. With this separation came greater flexibility in choices of languages, editors and operating systems. But with this flexibility also came a form of entropy — languages were all equally bad at talking to different file systems and editors were equally bad at supporting the quirks of different languages.

With the advent of yet more powerful personal computers which have several orders of magnitude, more backing store, main memory and processing power, it is once again possible to provide editors optimised to support individual languages. The computer industry is currently engaged in producing the successor to the personal computer, namely the *network* computer. These machines will have a common (browser) interface and will execute Java code on a "Java" processor [Way96]. It seems that the industry has moved almost full circle back to the point where a "personal" computer executes one, and only one, language with a standard interface (albeit a graphical one).

Whether executing on a Network or Personal computer, the RAD language should, of course, be supported by its own language-aware editor. Whilst this editor should provide the pretty-printing and syntax colouring / analysis, it could also provide much higher level support.

One way this might be achieved is through the use of hypertext editors where, for example, clicking on a variable name would jump to the declaration of that variable. Another alternative might be a folding editor, which behaves like an "outline" tool in a word processor, presenting successively lower levels of detail as the programmer requires. There does not seem to be an optimal paradigm for the novice programmer, as both these (and other) approaches have been evaluated ([MW],[SSSW85]) and found to cause confusion for novices.

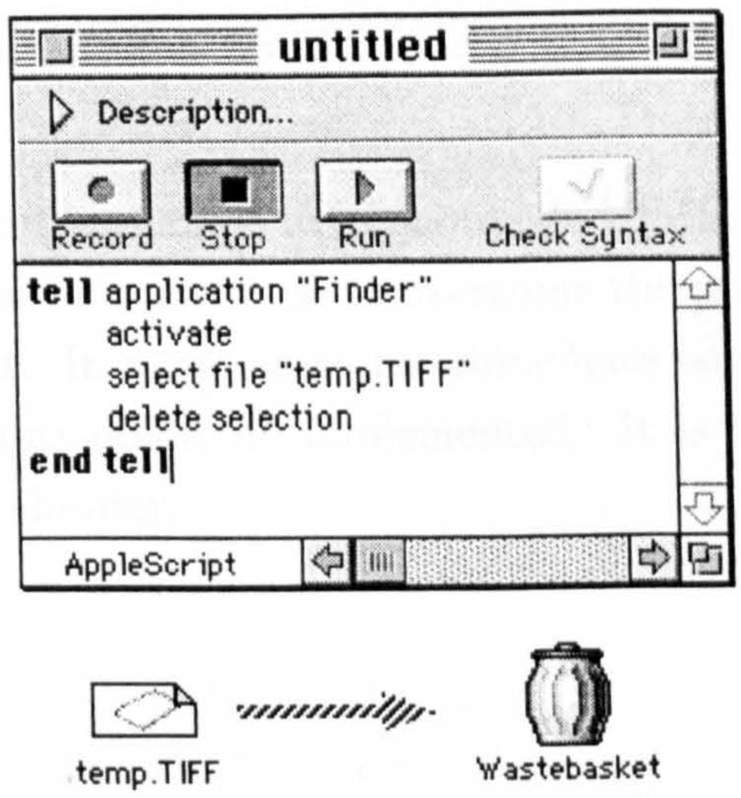


Figure 4.4: A simple macro is created by first pressing the "Record" button in the macro editor and then selecting a file and dragging it to the wastebasket.

4.9.9 Programming by example

Finally, a brief detour must be made to examine programming by example or, programming by inference systems. These were created to infer what a user was trying to achieve through watching that user interact with a direct manipulation interface. This has been a very active area of research producing many systems ([Mye86], [Cyp91] and [LK90]) and has been included as part of the latest (97) version of Microsoft Office [Mic97d].

Whilst this method does indeed free the programmer from learning any code, it would be naive to assume that it is possible to build a system capable of inferring any particular program the programmer might wish to express. Again, inferential programming introduces a plateau in the learning curve, when the tool is no longer able to support the programmer and they must resort to programming the behaviour explicitly. Inferential programming is best left in the domain of adaptive interfaces and automation of batch processes, where the problem space is tightly constrained[10].

Although not truly an inferential system, **macro recorders** can provide an interesting way to learn a programming language. One example of this is AppleScript, which allows the programmer to create macros, which can then be reviewed and edited. In this way, the programmer can be gently introduced to the concepts of the language and create working programs without initially needing to understand anything of the AppleScript language. For example, Figure 4.4 shows how to create a simple script without any explicit programming.

[10]Besides this, novices become suspicious of clandestine systems which seem to be making "intelligent" choices — for example, the author's father has not trusted a computer since one beat him playing Connect-4 in 1986.

4.10 Summary

Within this chapter an attempt was made to look outside UIMS and GUI research to evaluate other programming paradigms and use these to overcome the problems with RAD tools identified in the previous chapter. In some cases no immediate solution was found but insights were gained into how solutions might be implemented. It is these implementations which shall be pursued in the next chapter.

Chapter 5

Prototyping Ideas

5.1 Introduction

In the previous chapter an attempt was made to address the problems of RAD tools and show how some of these problems could be overcome through the re-application of previous research into language design. Other problems, however, were not so easily solved. Having discussed in the previous chapter how a solution might be found to these problems, this chapter will report on the implementation of these solutions. More specifically, this will include:

- The use of a functional language in programming GUI systems.
- How to merge the language and toolkit.
- Re-designing the interface creation environment.

5.2 Functional programming systems

The worked conducted within this thesis necessitated the implementation of a simple programming language. Because implementing parsers in a functional language is a fairly trivial task, and efficient execution was not crucial, it seemed appropriate to use a functional language for this implementation. This view was further reinforced through the reported success of functional languages in building prototype systems. Finally, the use of a particular functional language, Clean [PvE93], was further promoted through research links with the Clean

research group in Nijmegen. The experience gained in using Clean to implement this language provides useful insights into the suitability of functional languages to RAD applications.

5.2.1 Applying Clean

As expected, implementing the parser and lexical analyser in Clean was a relatively straightforward task. Using the parser builder functions described in [Rea89], different syntaxes could be easily explored. However, it was at this point that the language stopped being useful and actually became a hindrance to developing the system further.

Explicit typing

Functional languages have a very viscous type system. Clean is typical of other languages (Miranda, Haskell) in having a static type system which can infer types (using a Milner - Mycroft system) but can perform more comprehensive type checking if types are declared explicitly. Typing is strong, and any changes made in user defined types have huge repercussions throughout the entire program; for example, the development of the parser.

Parsing functions were designed to return either a success "ok" or a fail "fail" result. Trying to extend this so that "fail" could be subdivided into various error values required the rewriting of most of the parser, adding new pattern matching statements to each function to account for the new patterns generated by the extension to the type. If less explicit typing is used, then the role expressiveness of the language is lost and the initial reasons for using a functional language are eroded.

Unique typing

A problem specific to Clean is in the use of unique types to implement graphical user interface routines. Specifically, a unique type is defined as follows [PvE93]:

> A node **n** of a given graph **G** is unique with respect to node **m**, if **n** is only reachable from the root of **G** via **m** and there exists at most one path from **m** to **n**.

A variable declared of a unique type can only have a reference count of one. Because the value is guaranteed to have only one reference, it can be used to sneak in side effects (such

as destructive array updates) without losing referential transparency (i.e. no other reference can exist to that object, therefore an object referencing uniquely can guarantee that the referenced object cannot be changed by some other portion of code). Whilst this means that it is possible to produce a functional language which can implement a graphical user interface without the loss of referential transparency, it is at considerable cost to the programmer.

Firstly, it is not possible to infer the uniqueness of values, requiring that all unique values be explicitly declared. Secondly, unique types force the adoption of an unnatural programming style.

Consider the classic factorial function as defined in Clean (the "*" prefix on "*Int" is used to denote a unique value):

```
module fac

import StdEnv

fac::*Int -> Int
fac    1 = 1
fac    n = n * (fac(n-1))
```

As defined above, the factorial function will not work because the left hand side of the last line of the function contains two references to the unique value n. Therefore, n is not unique because in the evaluation of either reference to n a side effect could be invoked which would compromise referential transparency. To preserve this transparency using unique types, even the most fundamental functions must be re-written.

Clean implements its user interface environment as a unique type (called a "World") containing objects such as files, windows, buttons etc. (see Figure 5.1). The state of the interface can then be altered without the loss of referential transparency. Consequently, all functions written to manipulate the interface must be written to preserve its uniqueness. Although a conceptually coherent solution, the reason for using functional languages (i.e. their high role expressiveness) is lost. The language must be manipulated so that it can implement user interfaces, rather than supporting the interface semantics directly within the language.

5.2.2 Other functional language solutions

The problem of maintaining referential transparency is by no means unique to Clean. The communication channels presented by Singh [Sin91] again devolve responsibility for the user

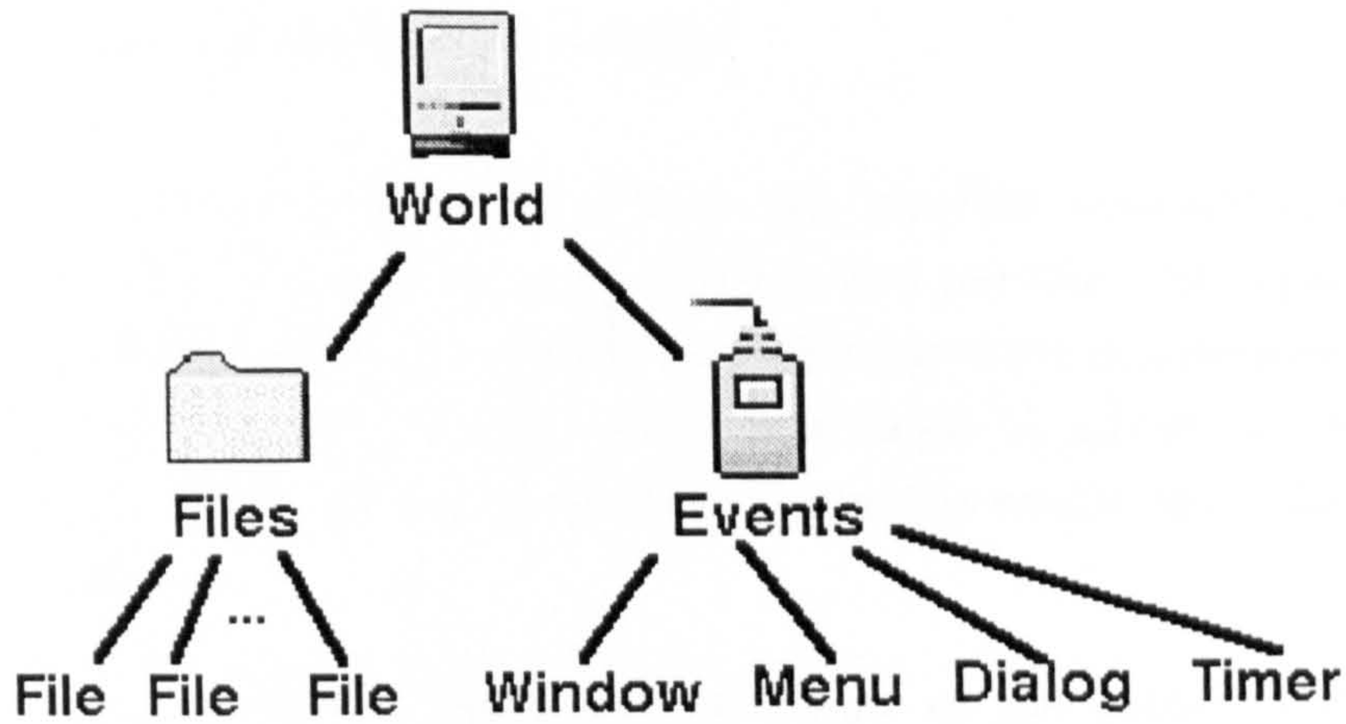


Figure 5.1: The contents of the Clean I/O "World."

interface to some new component of the language, which does not sit comfortably inside the functional programming heuristic. Like unique types, monads sit more comfortably within a functional language, but transformations on monads require code to be written in an imperative style (after all, the monad is essentially encapsulating state so that it cannot compromise referential transparency).

There would seem, therefore, to be little point in using a functional language for controlling graphical user interfaces *at all*, let alone at the novice programmer level. The functional programming community, however, are still producing innovative solutions, and refining those discussed above. It is hoped that somehow a language can integrate the concepts required for GUI programming into a functional language, without losing any of the role expressiveness and elegance which have fuelled their popularity. Programming graphical user interfaces in current functional languages is a task best left to students and researchers within the functional language community.

5.3 Language and toolkit merge

Perhaps the most commonly cited problem of user interface programming systems, the problem of integrating language and toolkit, must now be confronted. Two approaches were attempted, both of which are reported.

5.3.1 Finding the fundamental widget

In the previous chapter it was postulated that one possible abstraction for user interface widgets is to reduce the widget to its most fundamental particle (the button) and create all other widgets by adding properties to the button. Having made this decision, an attempt was made at providing a framework in which properties could be added to a button to convert it into other types of widget. It was hoped that properties would be a convenient, high level way to describe widgets [1].

First of all, to create a field, a "holds text" attribute of the button could be set to true. Furthermore, it is possible to add a "holds number" attribute to convert the widget to a scroll bar (or other appropriate representation). Other forms of button, for example a check box, can be created by adding a "toggle" attribute to a button.

Another area where adding properties would seem to work well is in programming the behaviour of widgets, setting the "dragable" or the "dropable" property of a widget. For example, movable dialog boxes could be distinguished from immobile ones simply by the use of the "dragable" attribute.

However, this approach rapidly becomes unworkable, for the following reasons:

- The "property matrix" became more complex than the original toolkit. For example, setting the "scroll" attribute of a field converts it into a scrolling field. So far, so good. Unsetting the "text" attribute converts the field into a scrolling window. If we now set, say, a "toggle" attribute, this creates a toggleable scrolling window; a non-existent widget! To prevent this type of problem happening would necessitate the construction of a widget dependency chart (as in Figure 5.2), showing what combinations of attributes can be used to create desired widgets. Even if widgets, or groups thereof, could be made mutually exclusive, this abstraction becomes confusing and bears little relevance to the problem domain.

- Not only is it the case that adding properties can produce non-existent widgets, it is also the case that there are widgets which cannot be described in terms of properties. For example, how would one begin to describe a menu bar? (A non-dragable family of scrolling fields?) Clearly the descriptions of widgets, in terms of their attributes, was more contrived than using the original toolkit.

- The focus of this approach is again too low level; the programmer spends time concentrating on the attributes of widgets and not on creating a coherent program.

[1] This hope was not unfounded, as a similar approach had been sucessfully implemented by Took [Too91]

	Text	Number	Scrolling	Toggleable
Field	√			
Check Box				√
Scrolling Field	√		√	
Scroll bar		√		
Popup	?	?	?	?
?	√		√	√

Figure 5.2: Whilst the first few lines of the matrix provide sensible descriptions, some widgets defy description (the Popup) and some collections of attributes describe nonsensical widgets (as in the last row of the matrix).

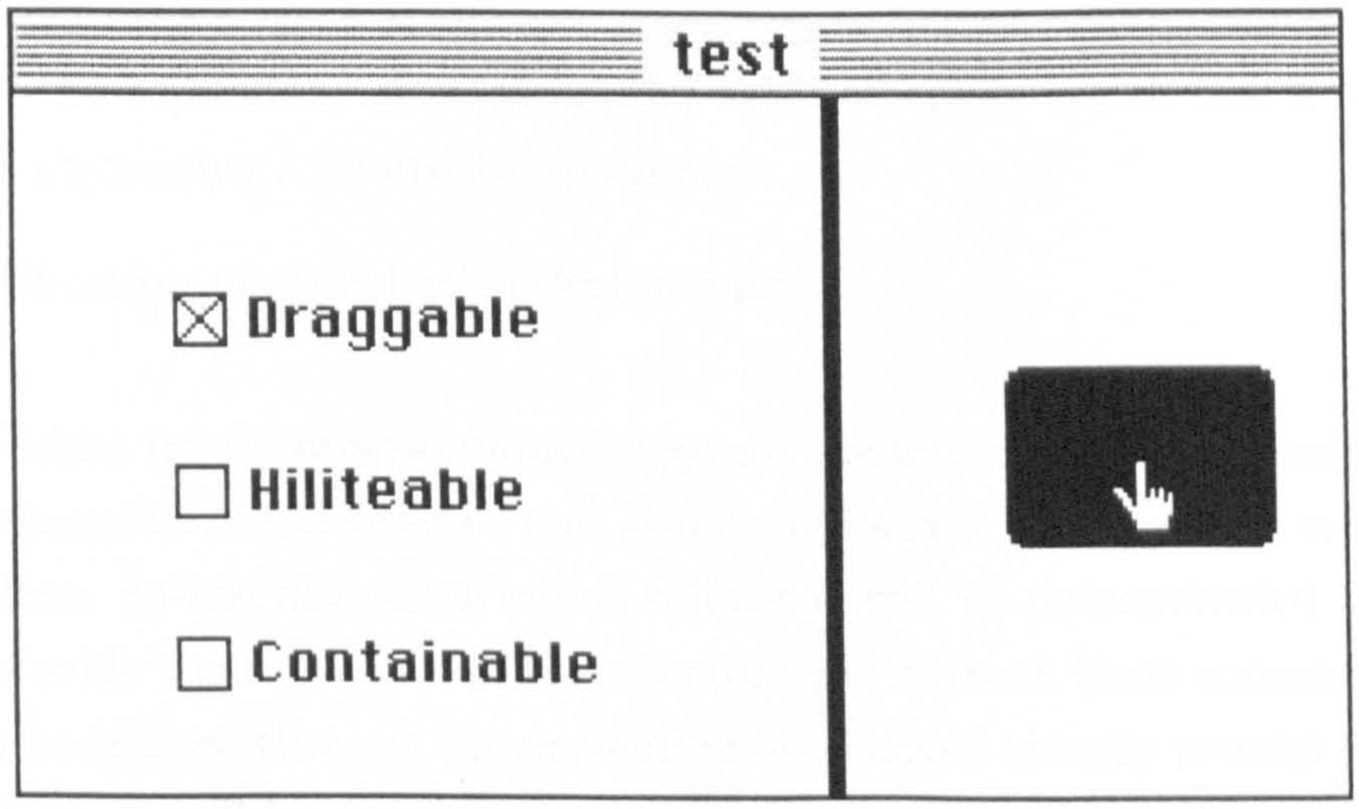


Figure 5.3: HyperCard prototypes were created to test different classification systems

Several classification schemes were built, using different properties, in order to see if a consistent classification could be found. These were tested with a variety of prototype systems (Figure 5.3), but no coherent set of properties could be found.

In reducing concepts and integrating the various components, it is important not to become overly parsimonious. As Green reports in [Gre90a], there are religious wars between those who favour large languages and those who favour the small, just as there are those who are pro CISC and those who are pro RISC. Both approaches contain value, and to overly simplify a language will inevitably lead to as much confusion as is caused by a large and rambling language. As Einstein is often quoted as saying:

> Everything should be as simple as possible, but no simpler.

The abstraction presented above would be analogous to removing all logic operators from a language except for NAND. Certainly, learning only one operator is initially less time

consuming, but rapidly leads to contrived solutions which require more effort in learning NAND combinations than in solving the actual problem.

5.3.2 Visualisation

The aim of amalgamating language and toolkit is to provide the novice programmer with a more gradual learning curve, so that two separate entities (the language and the toolkit) need not be understood before programming can commence. The work on ACE provided a starting place from where this might be accomplished by:

- Providing the separation of a (semantic) selector entity which can be visualised in a variety of ways using a (syntactic) presenter.
- The identification of some basic selector types.

To carry these ideas into a programming language would require the inclusion, in that language, of the semantics of the selector and also some form of visualisation routine to present it on the interface. In the discussion which follows it will be demonstrated that not only is it possible to provide the semantics of the selectors and present these successfully, but richer interaction can be created through the presenting of concepts already present in programming languages. Indeed, it will be demonstrated that most widgets found in current toolkits can be created by presenting programming language concepts and constructs.

To demonstrate this point, each widget will be examined to show that it can be created through the presentation of some part of a programming language [2].

Fields and Labels

Perhaps the simplest place to begin is with text boxes and labels as these are essentially strings displayed on the interface — assuming that the language has string types and string constants. The "presented" version of a string constant is a label and the "presented" version of a string variable is a text box (Figure 5.4).

---

[2]The majority of the ideas presented beneath were tested using a simple object based language implemented in Clean. As the concepts became more complex and the Clean routines less able to cope with them, other systems, including Java and HyperCard were used.

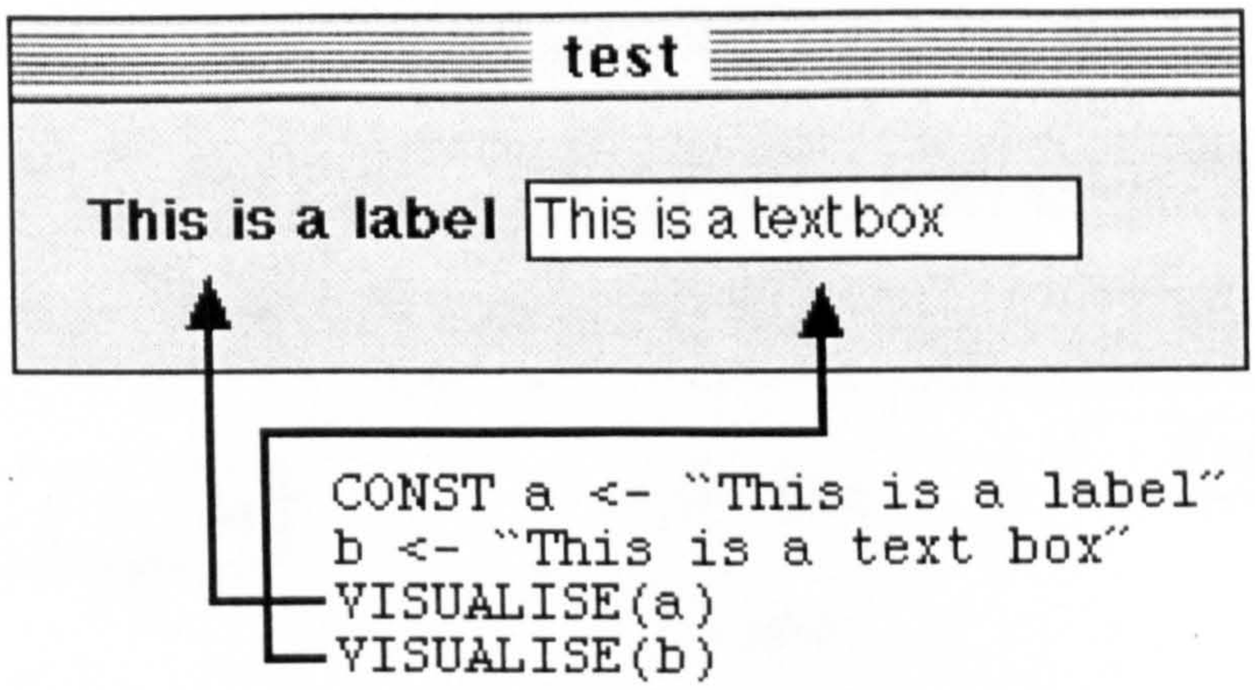


Figure 5.4: Visualising text variables and constants.

Radio and Check buttons

Radio buttons, like text boxes, already have a directly equivalent concept within programming languages, namely the enumerated type. Variables of a particular enumerated type can only hold one value from the set declared in the type definition. For example, in Pascal, the type `day` can be declared as:

```
TYPE
   day:(Mon,Tue,Wed,Thur,Fri,Sat,Sun);
```

Obviously, as radio buttons provide a mutually exclusive choice, they are semantically equivalent to the enumerated type. "`day`" can thus be visualised as in Figure 5.5.

The check box is a close relative of the radio button, but provides selection from a non-exclusive list. When the list contains only one item, then the check box becomes a visualisation of a boolean variable. If the list is longer, then it becomes necessary to introduce a new concept which we shall term the *variant enumerated (VE)* type.

Similar to the enumerated variables, variables declared of VE type can hold a list of values taken from the defined set, rather than the single value of the enumerated variable. For example, to implement a variable to hold the "style" of text in a word processor, the variable could take one or more values from the list (bold, italic, underline), as in Figure 5.6.

Whilst it so happens that these values are compatible, it can also be the case that selection of a particular value excludes other values. Again, taking font style as an example, the value "plain" would exclude other values. The declaration of a VE variable should therefore permit the exclusivity and inclusivity between values. The notation adopted uses boolean connectors, thus making the declaration of this style menu:

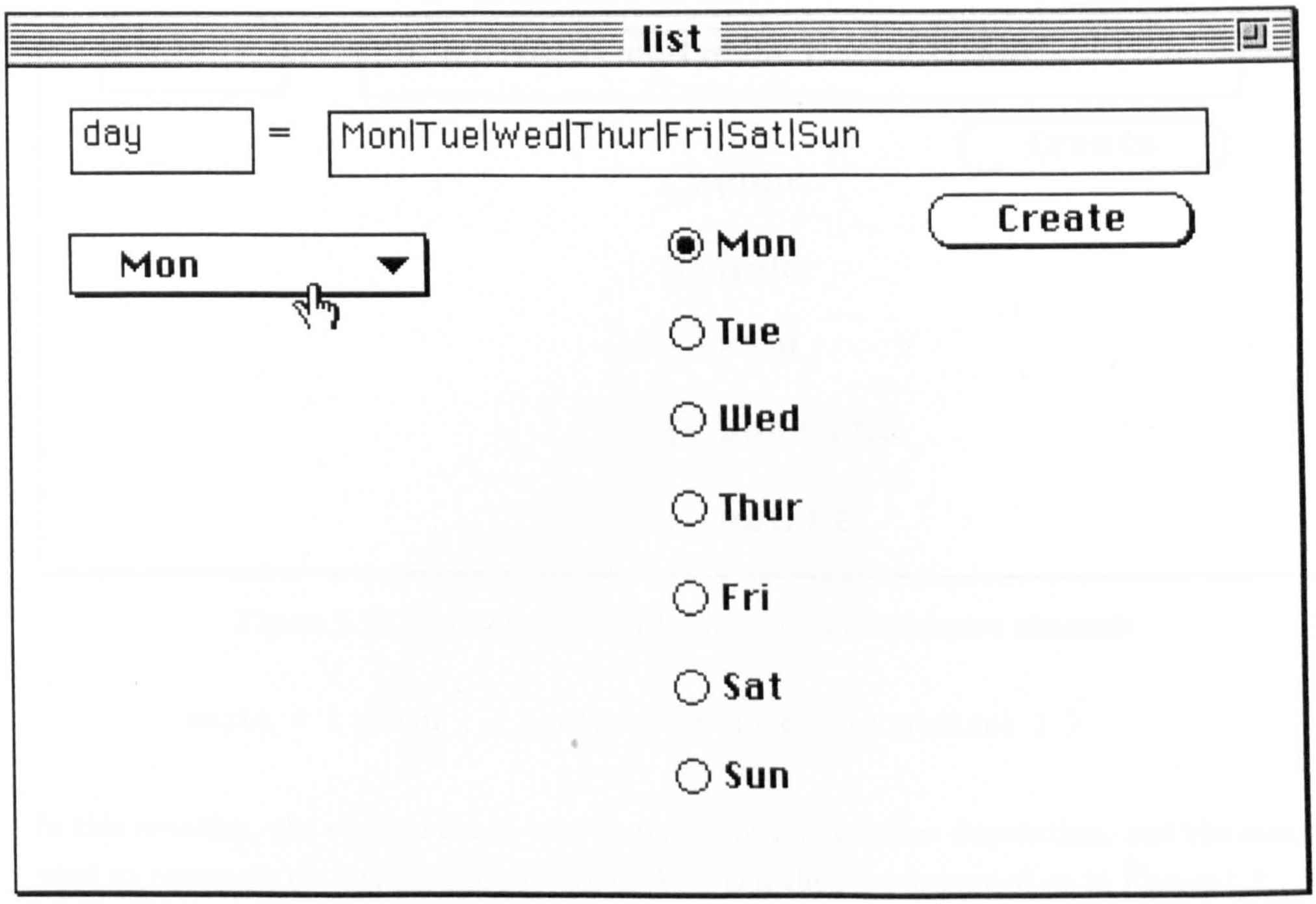


Figure 5.5: Enumerated type visualised as radio buttons, or a popup menu. (The "Create" button is used to generate the widgets once the enumerated variable has been specified.

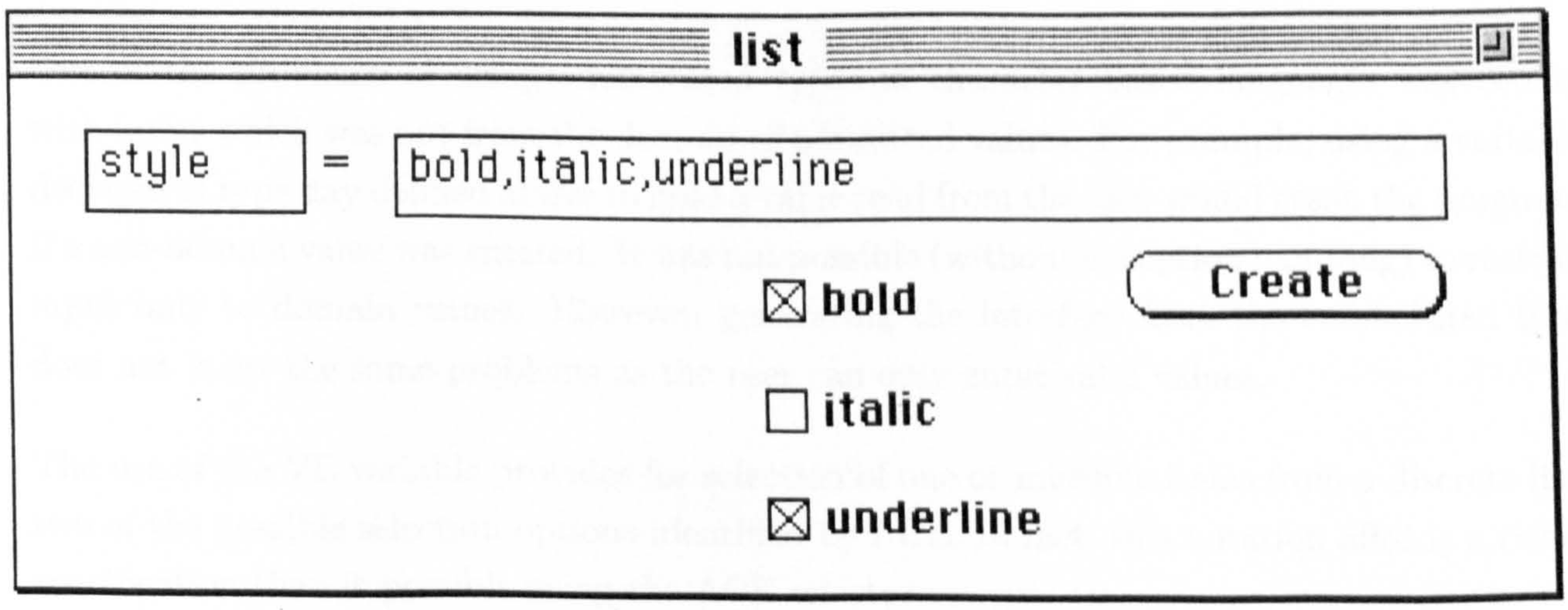


Figure 5.6: Selecting from a non-exclusive list

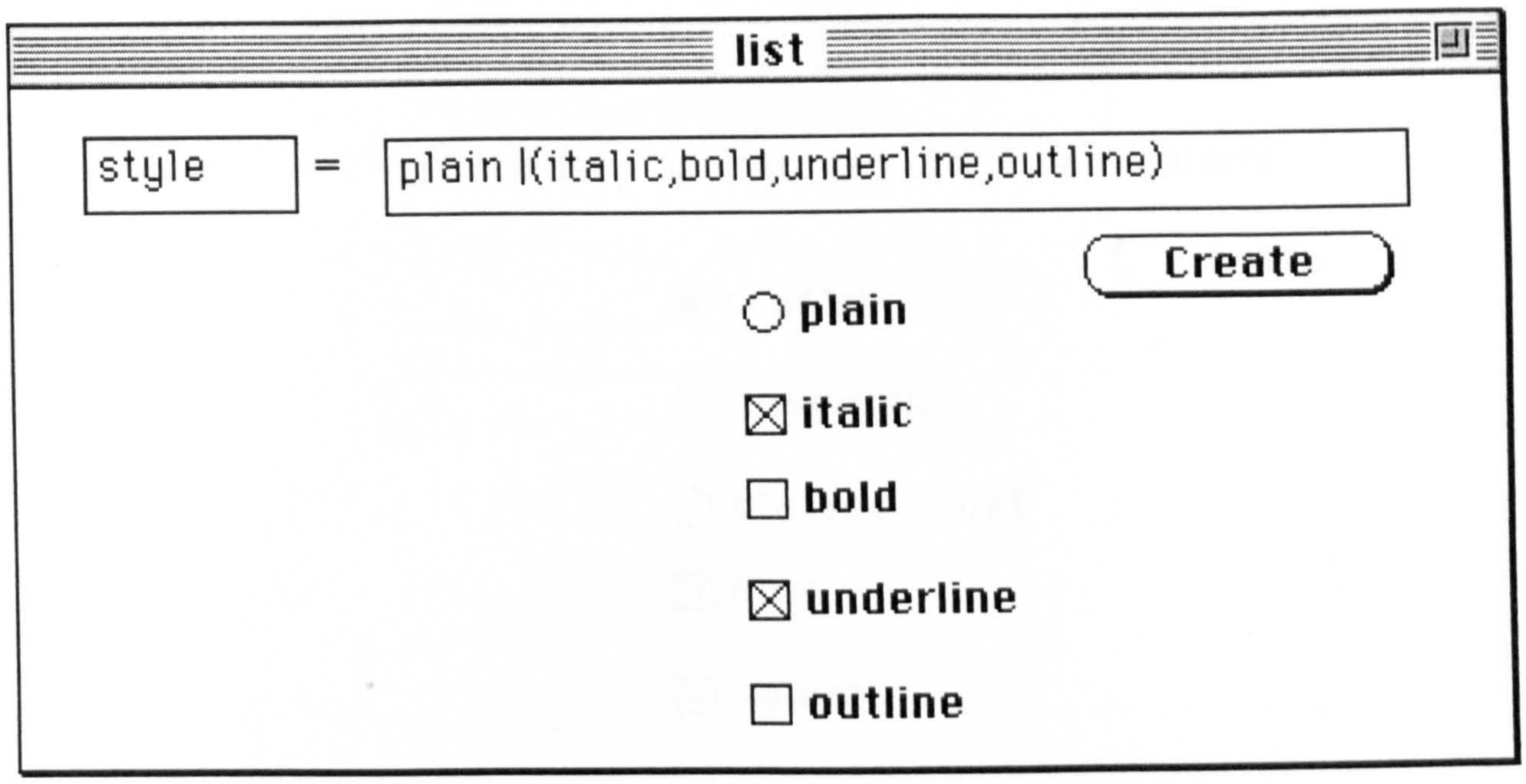


Figure 5.7: Creating a list with inclusive and exclusive elements

```
style = ( plain | ( italic,bold,underline,outline ) )
```

In this notation, the vertical bar is used to represent an exclusive disjunction, and the comma used to represent an inclusive conjunction. This can then be presented as in Figure 5.7.

N.B. An alternative notation was implemented using items listed as (*,*,*) as inclusive and those listed as [*,*,*] as being exclusive. This notation appeared less role expressive and experiments with novice programmers [Mar92] using Mathematica [Wol91] (which relies on three different types of parenthesis to denote three different concepts) would count in favour of the chosen notation.

One of the problems of using enumerated types in character based languages was coping with input which was not from the domain of permitted values. For example, using a variable declared of type `day` defined above to hold a value read from the user would crash the program, if a non-domain value was entered. It was not possible (without exception handling) to restrict input only to domain values. However, generating the interface from the enumerated type does not incur the same problems as the user can only enter valid values.

The use of the VE variable provides for selection of one or multiple items from a discrete list, two of the possible selection options identified by ACE. In fact, this notation affords a richer specification than is possible using the ACE selectors.

Presenting VE variables frees the programmer to concentrate on the problem domain, without being concerned with low level widget details, as the (slightly frivolous) Figure 5.8 demon-

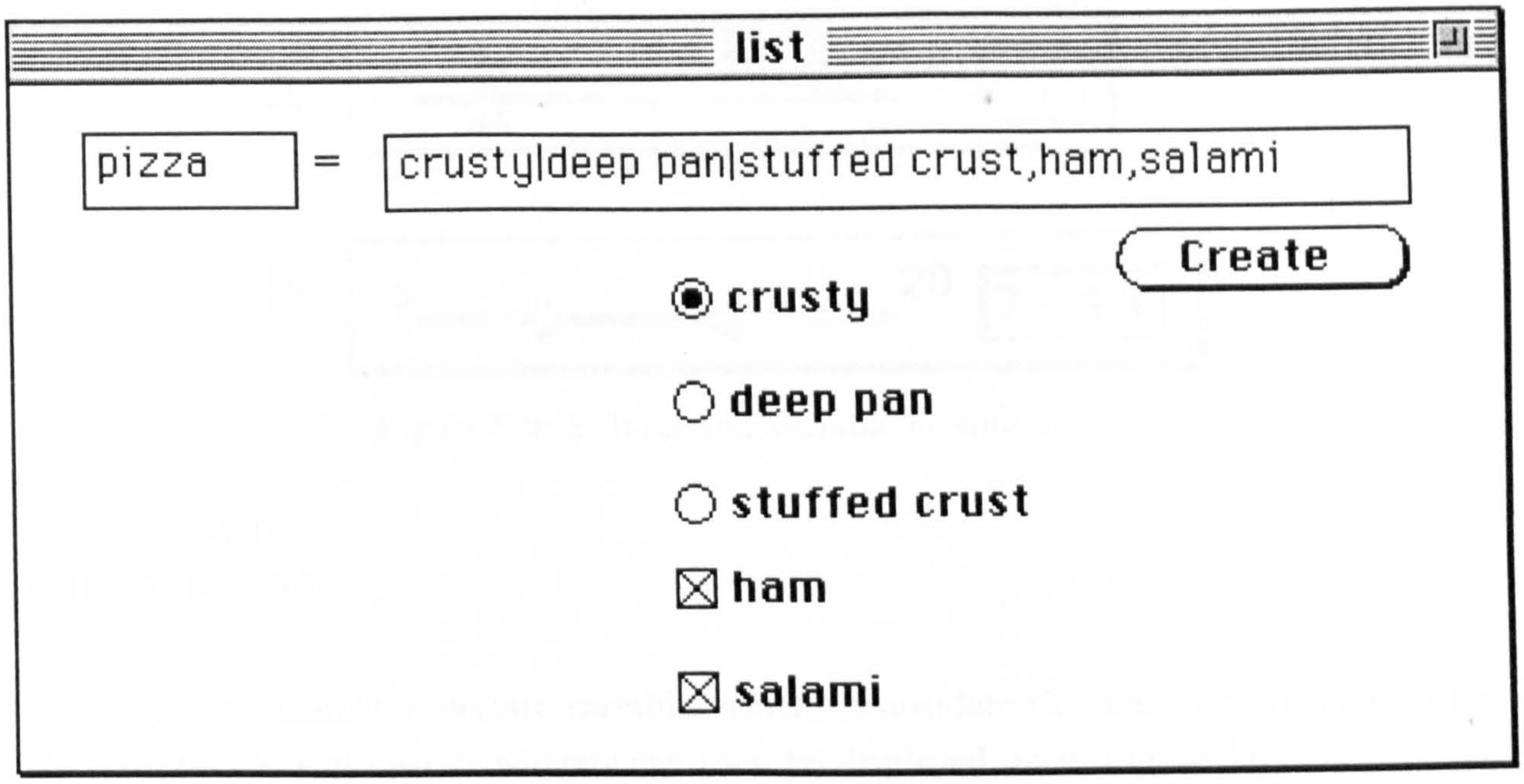


Figure 5.8: Enumerated types for pizza definition.

strates.

Scroll Bars and other sliders

By "presenting" any numeric variable, it is possible to create a scroll bar, selecting a value between zero and the maximum representable value. Practically, this is of little use, as the range is more tightly constrained. Furthermore, it must also be possible to select a sub-range of values (according to ACE) as well as the selection of a single value from a range.

Notations already exist in programming languages to specify the idea of a range, such as Pascal's range variables — `var a : 5..20;` declares a variable "a" which holds a single integer value in the range between five and twenty. Less clear is how to denote that "a" might hold a sub-range of values. It is tempting, as with the VE types described above, to use different parentheses to distinguish between single and sub-range selections. However, this is ultimately an unsatisfactory (and hardly a role expressive) solution. The notion adopted for the implementation is as follows ("a" can hold a single value, whilst "b" can hold a range):

```
a =   [5<-|->20]
b = [5<-<->->20]
```

This notation also allows the declaration of default values e.g.:

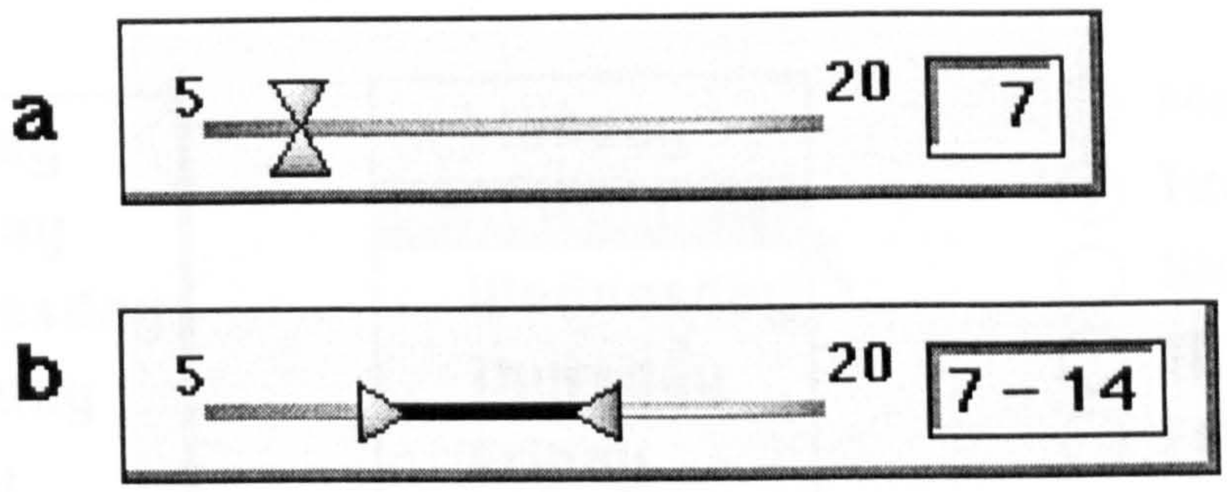


Figure 5.9: Sub-ranged variable as slide bar.

```
a =  [5<-7->20]
b = [5<-7-14->20]
```

In this way it is possible to declare variables which encapsulate the semantics of the final two ACE "selectors." Appropriate widgets can then be displayed, as in Figure 5.9.

Obviously this could be extended with steps of increment commands and commands to alter the maximum and minimum values of the range.

Selection Menus

Menus are one of the earliest forms of computer interface. Essentially, the menu permits selection from a list of options — there are, however, two different types of selection.

The first type of menu selection (which shall be termed a "command" menu) invokes some command or executes some piece of code. For example, the Macintosh "Apple" menu contains various applications and desk accessories which will launch when selected from the menu.

The second form is the "selection menu," which is a more passive type of menu, whereby the selection of an item has no direct subsequent effect. With this type of menu, the user is performing a selection which will be used in subsequent processing — the action of selection does not directly invoke any method.

The first type, or "command" menu will be considered in the next section. The "selection" menu will be considered now, however, as it is no more than an alternative rendering of the enumerated type.

Enumerated types which are entirely mutually exclusive (i.e. could be presented as radio buttons) can also be displayed as a pop-up menu or a "tick" menu (where choice is indicated by selecting a menu item, toggling a tick beside the entry). All forms are semantically equivalent (selecting one discrete value from a list) as can be seen in Figure 5.10.

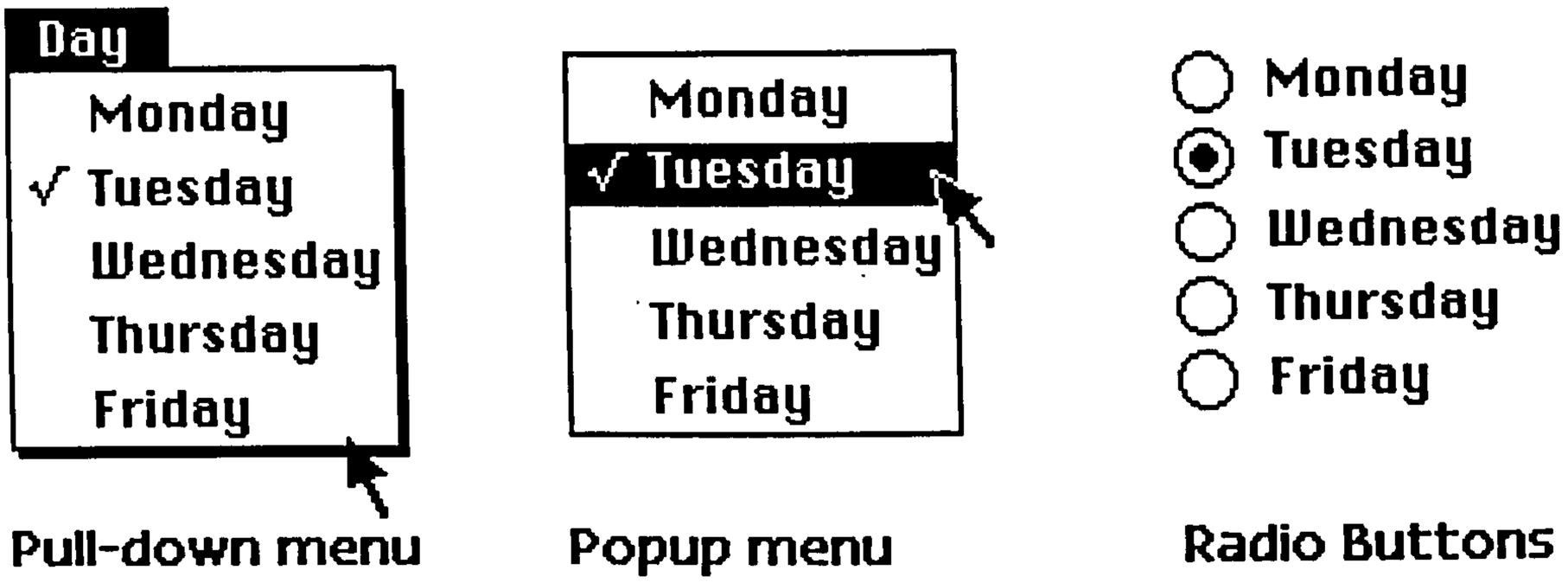


Figure 5.10: Three different presentations of the same concept

The other case permits multiple selections from a list of discrete values. As demonstrated before, such a selection can be presented as a mixture of radio buttons and check boxes, but can also be presented as a "tick" menu (in either drop-down or scrolling list form, but not as a pop-up).

Various notations have been developed for the specification of menus. Perhaps the most famous of these is Lean Cuisine [AS89]. The notation specified here is as expressive as the Lean Cuisine notation, but has the benefit of being integrated into the language. The designer does not need a separate tool to view this design and the specification can be used directly for further application development.

One addition to the Lean Cuisine notation is the use of the '*' character to denote the default value of a menu. It would seem useful to adopt this approach within this notation, which would provide for the initialisation of VE variables. Again, taking the style menu, it could be declared thus (the effect being to initialise "style" to "plain"):

```
style = ( plain* | ( italic,bold,underline,outline ) )
```

5.3.3 Making a language

Thus far, a description has been made of how the visualisation of individual concepts produces interface widgets. If these concepts are to be used in creating an interface, then they must be integrated into a language and used to create complete applications, which will require the grouping of widgets into dialog boxes, frames and windows. The sample language created to test these concepts works as follows:

Variables were declared as normal, making the program appear identical to any other third generation language:

```
loop = 0
while loop < 10
loop <- loop + 1
endwhile
```

By adding the extra statement `visualise loop` after the declaration, a window automatically appears on the screen showing the value of `loop`. Also, in the object store, the system automatically creates rendering attributes for `loop` such as "height, width, etc." These will have default values, which the programmer (once they learn that these attributes exist) is free to alter — for example `loop.font <- ''Times''`. The language also allows the programmer to add event handlers to objects — for example `loop.click=(loop.font <- ''Times'')`. It was not within the language's capability, however, to visualise any variables created as part of the event handler for a widget — this was a design decision, as visualising a part of a widget could have no sensible interpretation.

In effect there is no distinction between base type variables and other objects within the language, as the base type variables have methods and attributes other than a simple "value". Within the language there was no explicit notion of a class: new instances of an object were created by an assignment statement, copying the attributes and methods of one object to another. This is identical to the creation of classes in current RAD systems, where inheritance is static; ideas about the provision of dynamic inheritance are discussed later.

There must, of course, be some sort of lexical scope in which a variable / object can be declared. Within the language, these were termed a "part" and contained both variables and methods. The idea of a "part" is to provide some form of scope within the language to group variables and methods (i.e. a general purpose object.) Within the implementation there was no class mechanism — creation of a new part was made by snapshotting some other object.

Like the language variables, parts can also be visualised. Depending upon the nature of the objects the part encapsulates, the part can be rendered as a dialog box, window, or even a command button.

Command buttons and menus

One way to think of command buttons and command menus is as subroutine calls, invoking some piece of code. In the context of an object-oriented language, these pieces of code are the

methods of objects. How then is it possible to "visualise" a method and make it accessible to the user?

The methods of the visualised variables are already accessible from the interface, as they form part of the behaviour of the visualised widget. If a "part" method is declared as responding to some external event (e.g. DoubleClick, MouseEnter) then it must obviously be visualised as some form of interface device so that the method can be called. If a "part" contains only one presentable method and no presented attributes, then that object will be presentable as a command button.

If there are several "parts" declared inside another "part", then the higher level object can be presented as either a group of command buttons or a menu.

By presenting menus in this way, it is impossible to mix menu types; i.e. a single menu cannot contain both passive and active elements. Rather than being a limitation, it may be that forcing the separation of active and passive menu elements will create a better structured and more coherent interface. In that respect, this separation can be considered a benefit.

If an object containing only one "click" type method and containing no presented variables is presented as a command button, what of objects which contain other presentable methods and presentable variables?

Other buttons and group widgets

If a part that contains a "click" method, and nothing else, is presented as a command button, then a part with more than one method (at least one of which responds to a mouse event) becomes a general purpose button — identical to the "button" type in HyperCard. Buttons of this type not only respond to the click method, but also to double click, mouse enter, mouse leave etc. and can be programmed to perform a custom response.

The inclusion of parts also makes it possible for the programmer to create customised widgets. A part, which contains event handlers and a single visual "picture" variable, could be used to implement new forms of button.

Parts which contained other visualised variables (not just multiple methods and a picture control), however, were presented as dialog boxes containing those variables. For example, consider the following program written to implement a simple desktop calculator:

```
Part calc
```

```
{
   screen <- 0
   memory <- 0

   Part one
   {
      on click
      {
         screen <- (screen * 10)
         screen <- screen + 1
      }
}

Part two
   :
   :
Part M
{
      on click
      {
         memory <- screen
      }
   }

   on load
   {
      visualise (one, two, ..., M)
      visualise (screen)
    }
}
```

The parts (one, two, M etc.) are the buttons of the calculator, which are visualised as command buttons. The screen variable is visualised to provide the screen. The "calc" part is then visualised automatically by the system. "Calc" may contain methods (`on load`) and variables (`memory`) which are not visual and may not be accessed by the application's user.

By visualising the lexical structure in this way, a direct correspondence is provided between the structure of the interface as it appears to the end user and the application programmer. This provides benefits to the novice programmer as the hierarchical abstractions of the language

are identical to those of its visual appearance.

Another benefit of having a symmetrical structure is in supporting end user customisation of the application — users can find the property or method they wish to amend directly through the interface. In the next section, an interface builder metaphor is presented which would allow end users to exploit this symmetrical structure.

Further use of parts

The implementation of this language, as it stands, only supports the dialog type of window. This is simply an implementation restriction. Had a more flexible environment (more flexible than Clean, that is) been used for implementation, then windows and other forms of dialog box could have been made available to the programmer. Also, in the current implementation, communication between parts is made through shared variables.

5.3.4 Automatic presentation and layout

Throughout this discussion, no mention has been made as to how each of the abstractions is presented. To a large extent, this process can be automated, freeing the novice from having to make this choice. Papers such as [dBFM92] and more recently [BV94] discuss schemes for widget choice based on data type and data size. This scheme can automatically choose from amongst 320 different widgets. The automatic selection of widgets can quite competently be handled automatically, but how can the concept of visualisation be built in to a language?

One possible approach is to borrow and adapt the concept of "the root of persistence" from persistent languages such as Napier-88 [Gro97b]. Within this class of language, any variable which is to be made persistent is attached to a root of persistence, producing a tree-like structure, so that when a parent is detached from the root, its children are also automatically detached.

Whilst it is possible to provide a "root of presentation" within a GUI programming language, there is no need for declaring an explicit structure. In the scheme described above, presentable objects are declared within other grouping objects. If these lower level objects are presented, then all enclosing structures must also be presented. Conversely, if an enclosing structure is hidden (or un-presented), then all enclosed objects must also be un-presented. Consequently, the presentation dependencies can be treated as implicit. In the final implementation then, the language contains a "visualise" keyword which automatically selects the most appropriate widget.

It can be imagined, however, that as the programmer gains in experience, they would want specify their own widget. For example, `visualise day = (Mon|Tue|Wed)` may cause the system to select a group of radio buttons — the programmer should also be allowed to specify the widget they desire `visualise day = (Mon|Tue|Wed) as popup`. Finally, the language should allow variables to be declared as visual; e.g. `visual var x = (1<-<->->50)`.

There remains a tension between inferring the most appropriate widget and allowing the programmer to explicitly define the widget. Were the definition of "x" above to be restated as: `visual var x = (1<-<->->50) as scrollbar` "x" could not be sensibly presented as a scrollbar. If "x" were to be presented as a scroll bar, then its definition would need to be altered to fit the widget. In cases where a widget can be used to present more than one type of value, such a redefinition would be dangerous. A better solution is to give precedence to the automatic selection scheme. If it overrides the programmer's choice of widget, then they are immediately made aware of this alteration as the interface appears differently than they had anticipated.

Not only must the widget choice be automated, but a default location, within a particular window or dialog box, must be decided upon. The development of device independent interface programming languages, such as Java, provides mechanisms by which this can be accomplished. Rather than requiring absolute co-ordinates, Java adopts the idea of a "layout manager" which will attempt to place widgets in the most appropriate place. Although this does not always produce the desired result, it provides a starting point from which the programmer can begin to refine the interface. Figure 5.11 illustrates the use of a layout manager in a Java implementation of the ideas presented above.

Again, the adoption of this scheme will allow the novice programmer to progress safely until they can make sensible, better informed decisions about explicitly specifying the interface structure.

5.3.5 Improvements

During the implementation of this language, several ways of improving this, and other RAD languages became apparent:

- **High level events:** Most events produced by the operating system are related to the domain of the device producing the event. Whilst this provides a general purpose mechanism for processing events, it is hardly orientated to the problem domain. With the introduction of visualised variables, it becomes possible to introduce a higher level, "change" event which is generated when the value of a particular variable is altered.

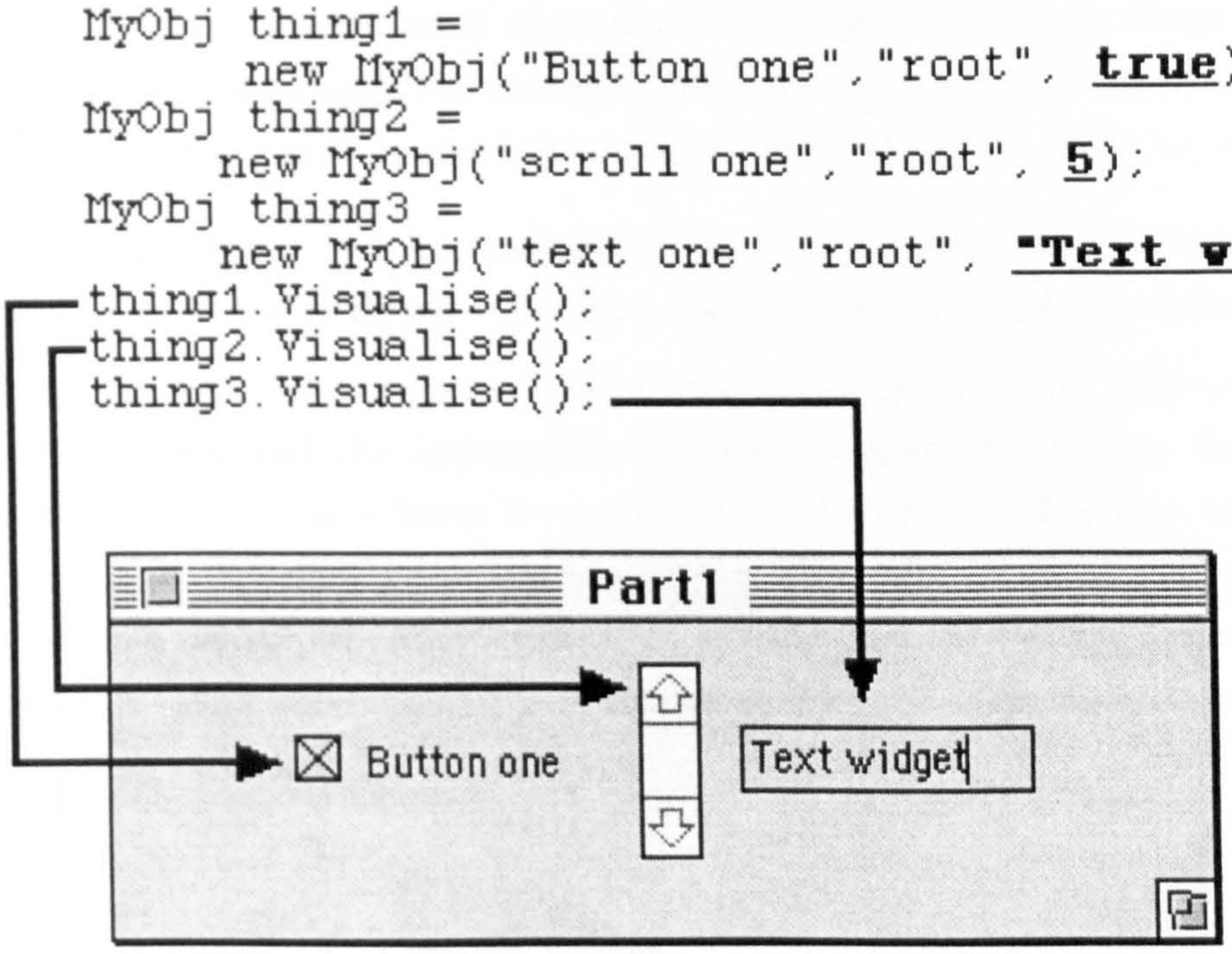


Figure 5.11: The three values in bold underline are visualised by the system, which chooses the most appropriate widget and location.

Because the widget in question is a visualised variable, the change may be initiated either by the user (in the form of a mouse click or key press) or may be modified by an assignment statement within the code. The programmer need not concern themselves with why the variable changed, they can simply add a handler to an enclosing part, which will effectively watch for a variable to change. This type of event makes for an interesting "equal opportunity" [Thi90] programming heuristic where the interface user and the program code can be thought of as equal agents performing transformations across the state of the program.

- **Dynamic inheritance:** In most third generation languages a pointer is used to make the distinction between creating a static copy of an object or variable, or simply declaring a new reference to the original object. HyperCard and Visual Basic do not provide any equivalent to the pointer: all references, as it were, are "by-value." This concept, deemed too hard for the novice programmer, is present in most Macintosh applications in the form of "Publish and Subscribe" which is a form of dynamic copy and paste — if the source of the "copy" is altered, the "pasted" version is automatically updated. Also present is the notion of Alias (on the Macintosh) and Shortcut (for Windows), whereby the user creates a reference to a file, rather than a file copy. Surely either one of these is an ideal metaphor to introduce the novice programmer to the concept of a referenced object.

Furthermore, this notion of aliasing can be used to introduce visual constraint programming to the language. For example, the following code would set the left edge of widget one to be five pixels to the left of widget two: `widget1.left <- widget2.left + 5`.

- **Type system:** The implemented system is typeless, with automatic coercion rules, similar to those of HyperTalk. How then can strict typing be enforced?
  One possible way is to treat a type mis-match as an event. The event is caught at the system level and the appropriate handler applied. Of course, the programmer could catch this event at a lower level and apply their own conversion routines[3]. If the event handler is written in the base language, then the programmer may even modify the standard behaviour. Furthermore, the language could be supplied with various standard type checking behaviours which the programmer could select depending on the type of checking required.

5.3.6 Symmetrical Visualisation

One of the problems with using properties as an abstraction of widgets is that:

- There are widgets which cannot be described sensibly by a set of properties.
- There are sets of properties which describe non-existent widgets.

Table 5.1, describing the solution proposed in the preceding section, shows that those widgets common to most toolkits can be described, and that visualising language elements produces a sensible result.

5.4 Friendlier Environments

One of the supposed reasons for the success of the GUI is the use of metaphor and thus exploitation of users' knowledge. The benefits and shortfallings of the use of metaphors in interface design have been widely discussed, but the RAD tool introduces a design behaviour to widgets entirely unlike their real world counterparts. Is it possible, then to provide a metaphorical equivalent for this design time behaviour?

[3] If these conversion routines contain a mismatch error, then the handlers at the system level will still catch the mismatch.

Widget	*Language Concept*
Field	String variable
Label	Constant
Scroll bar	Numeric range variable
Single check box	Boolean variable
Multiple check box	Varient enumerated variable
Radio buttons	Varient enumerated variable
Popup menu	Varient enumerated variable
Passive menu	Varient enumerated variable
Command button	Part with one click method
Active menu	Many parts with one click method
Dialog Box	Part containing visual variables

Table 5.1:

If a user wishes to repair or alter an object in the real world (say their personal computer), they immediately examine the casing in an effort to find a screw, or similar fastener, which allows them to take the object apart. By removing all the external fastenings, they have access to the mother board and all the sub assemblies inside the machine. These, in turn, have more fastenings which can be removed until the target component can be found. Might this same approach be applied to editing and executing a direct manipulation interface?

5.4.1 A new metaphor

Imagine an interface which is in "execute" mode, with a single screw affordance visible on its surface. Clicking the pointer on the screw widget causes the screw to revolve and pop out of it's hole. The surface of the interface slides back to reveal screws on individual widgets (such as menus and frames) and also a tray of available new parts. Clicking the screw of an individual widget reveals a property sheet for that widget, allowing the user to change its various properties. All other widgets on the interface will continue to retain their execute behaviour. Clicking on a unscrewed widget allows its position to be altered by dragging it to a new location. Once alterations to a widget are complete, it can be screwed back in to the interface. Such an interface was implemented to replace the standard tool set in HyperCard, a sample interaction being described in Figure 5.12.

The interface metaphor discussed is essentially a blending of a mode-per-object and an affordance solution. By providing a single, generic affordance on each widget, potential cluttering is eliminated. The affordance can then control the mode of the widget, denoted by the screw state (in or out). This mixture provides an environment which is consistent with user's real world experience.

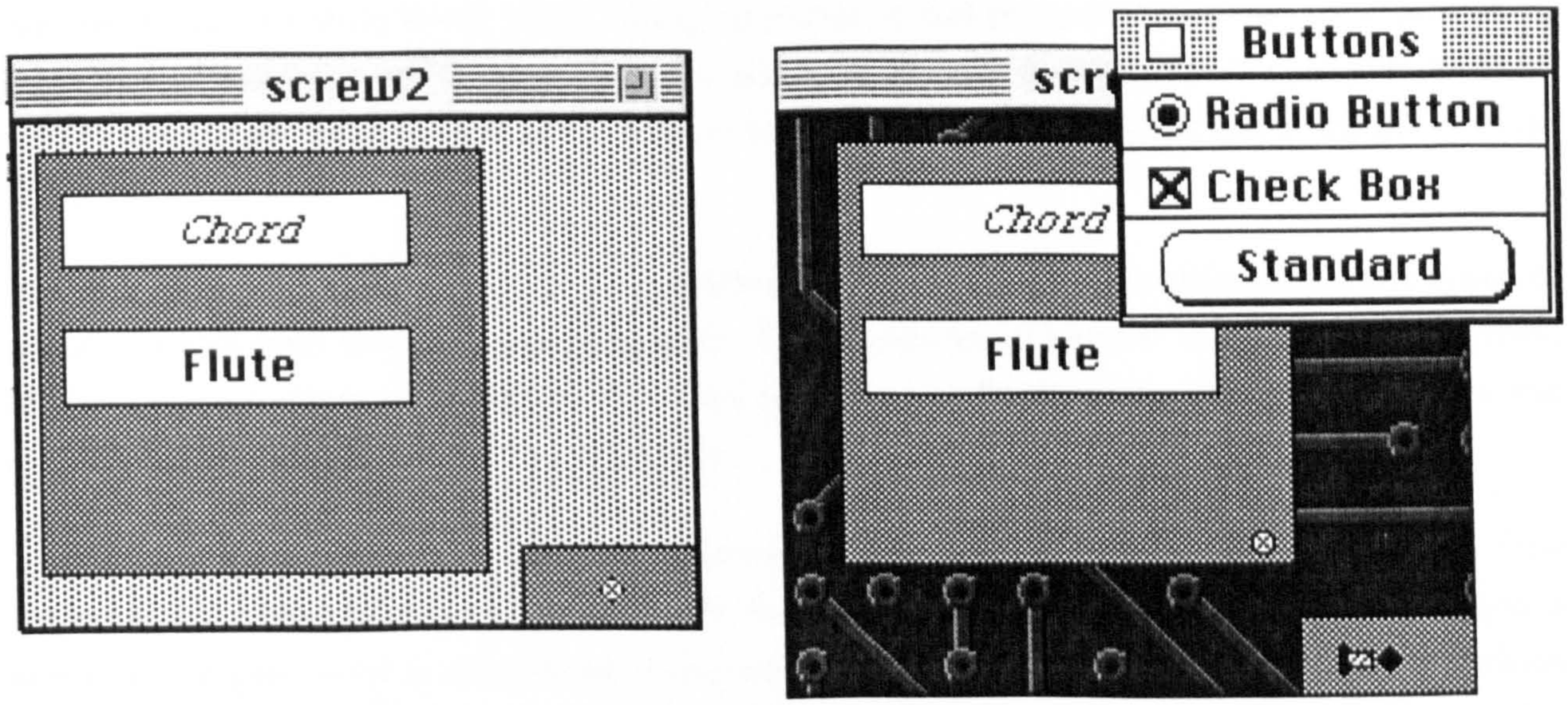


Figure 5.12: In the first image, no editing is taking place and the screw in the bottom right hand corner is inserted. When this screw is removed (as in the second image), the "surface" is removed and a part tray appears. Screws then appear on the individual objects which, when removed (as in the third image) present a dialog which allows the widget to be modified.

The other benefit of this approach is to completely eliminate the whole notion of design and execute modes, the mode residing with a particular widget. By integrating the modes in this way, it is also possible to provide a simple route into end user customisation of applications.

By providing a range of fastenings from a knurled nut through to an anti-vandal torx screw, a variety of customisation levels (from changing colour to full programming) can be provided. If attributes of widgets could be given priority levels (as discussed above) these could be assigned to different grades of tool, just as the higher levels of user in HyperCard have control of more of the environment.

Not only does this allow end users to customise applications, but providing different types of screw can also help the novice programmer. The provision of a screw encourages exploration of the customisation mechanisms, but also provides feedback as to the complexity of the customisation they are about to undertake.

Like all metaphors, however, the virtual screw deviates from its real world counterpart. One improvement that the virtual screw has is that all sub-level screws can be replaced when a screw at a higher level is refastened. Conversely, it is possible to provide a command which removes all screws. With these two options, it is possible to make this approach behave like a system with distinct design and execute modes.

As this idea was being developed, the author was given various "useful" suggestions as to how the realism of the metaphor might be improved. These included: have springs fly out to be irretrievably lost whenever the lid is removed; having screws randomly disappear; leaving several screws behind when the lid is closed. It would be unwise to include these features in any system which a novice might use!

There are some other ways, however, in which the realism of the metaphor might be usefully improved. For example, a property sheet is hardly a familiar concept. One approach examined was to replace the property sheet with special customisation widgets (such as potentiometers) as found inside real electronic equipment. This approach started to become ridiculous as customisation widgets were sought for altering attributes such as "width" and "bevel depth." Like all metaphors, either electronic or literary, the stretching must stop somewhere.

5.5 Summary

Within this chapter the use of functional programming for GUI applications was rejected. Also rejected was the abstraction of describing widgets by properties. The abstraction which was promoted for integrating toolkit and language, however, was that of visualising language

constructs. A new metaphor was also proposed for eliminating design and execute modes. In the next chapter an examination will be made to determine if these new proposals provide benefits over existing systems.

Chapter 6

Conclusion, Appraisal and Further Work

6.1 Chapter review

We have endeavoured to discover the programming solutions available to novice programmers wishing to implement their own GUI applications. The first chapter defined the programmers in which we are interested, the types of task they wish to complete and the types of tools they might use. Reviewing this situation revealed the need for a graphical interface programming language. Apparent lack of research in this area was cited as the main reason for undertaking the work presented here.

6.1.1 Elusive literature — defining a framework

Due to the lack of research discussed in chapter one, the second chapter was not the typical literature review one might expect to see in most theses. Instead, because this is such an implementation driven[1] field, a survey was made of the types of tool available to the GUI programmer and criteria were set for identifying and defining RAD tools — the class of tool with which this thesis is concerned. The documents reviewed did not themselves attempt a definite classification of RAD tools and the classification presented here will help overcome some of the confusion that has arisen through the current fad of labelling RAD tools as

[1] i.e. the output from most research in this area is some sort of programming system, rather than detailed papers on design rationale.

"Visual" languages.

Having defined how to identify a RAD tool, the next task undertaken was providing a framework for the comparison of these tools. To that end, a blend of criteria was created from reviewing comparisons of the RAD tool's ancestors, then measuring how well these disparate elements integrated. Using this framework, a comparison was made between two of the more popular RAD tools — HyperCard and Visual Basic.

When this work was started, HyperCard and Visual Basic were the most popular RAD tools. Little in computing science stays static, however, and the marginalisation of Apple products, along with an increase in the popularity of Internet scripting languages mean that, were the work to be started now, different systems would have been chosen! Whilst this may weaken the argument of the thesis to some minds, because HyperCard and Visual Basic were from the first generation of RAD tools, they are more likely to manifest more weaknesses than subsequent generations of systems. It is through the exploration of these weaknesses that the clues were found to guide the work presented here.

The second chapter resulted, therefore, in the following list of attributes and for each attribute, an explanation of how it was implemented in HyperCard and Visual Basic:

- Environment design and execute modes
- Code translation
- Extending widgets
- Platform support
- End user customisation
- Program editors
- Language structure
- Language syntax
- Data types
- Data structures
- Data persistency
- Multimedia data
- Error handling

One further attribute was dealt with — that of the integration between language, toolkit and environment. To aid comparison, a diagramatic notation was developed to represent this integration visually.

6.1.2 The good, the bad and the RAD

The list generated in chapter two merely listed the implementations of each attribute, but made no comment on whether a particular implementation was better than any other. Before that type of comment could be made, it was necessary to gain some understanding of what constituted a better, or worse, implementation.

To that end, a review was made of work in UIMS evaluation. Most of this work was surprisingly disappointing as reports dealt with outlining the positive and negative attributes *of one particular system.* This did not permit direct abstraction of the value judgements to any another system. This required the analysis of the common failings and successes in these systems to produce some root causes:

- **Badness** Seemed to originate from using old text-based technology in a graphical environment.
- **Goodness** Seemed to come from having concepts within the language which mirrored those in the graphical environment.

Besides these points taken from the literature on UIMS, other literature was consulted which related only to the graphical run time system of the RAD tool. Whilst this component of the RAD tool is similar to most direct manipulation interfaces, it exhibits unique problems in the way it presents the different design and execution modes to the end user. These separate modes can cause confusion to the novice programmer and new systems should therefore seek to eradicate them.

Finally, the integration of language and toolkit was considered. Because of the adhoc way in which these systems were created, by combining a language and a graphical environment, the novice programmer is forced to learn each component separately. Whilst users could learn the graphical environment initially, before they could add behaviour to the interface, they had to learn the underlying programming language. Learning the language initially required an investment of time which yielded no immediate return.

Having identified, in generic terms, potential sources of "goodness" and "badness" in RAD design, each of the RAD system attributes (identified in chapter two) was examined. In-

specting the HyperCard and Visual Basic implementations, with respect to the goodness and badness guidelines, produced a list of suggestions for how each attribute should appear in future RAD systems.

Having derived a list of ideal properties for each attribute of a RAD system, the problem remained of how to actually implement a system which realised these ideals.

6.1.3 Moving forward — which way now?

Before a complete system was designed, inspiration was sought on how one might undertake this task successfully. To that end, design principles from third generation languaes (e.g. Principle of Correspondence) were re-interpreted to see if they were still meaningful for a language which included a graphical component. These principles suggested that the following be true of any new system:

- Language should support multimedia types
- The language environment should be implemented in the language itself
- Proper interface abstractions should exist within the language
- References should be bound using visual scope
- The environment should be extensible by visual components which have the same rights as the components which are part of the original distribution

Every item (but one) in the list above provides explicit guidance in how to implement any new RAD system. The one, less explicit, principle is that the language should contain proper interface abstractions. This also mirrors the design principle from chapter two that within successful UIMS, the language structures should mirror those in the environment. Discovering what these abstractions might be is discussed later in the text.

Besides examining ideas from research in third generation language implementation, cognitive dimensions (viscosity, role-expressiveness etc.) and surveys of trainee programmers (negative and positive engagement) were sought in order to guide the design of the final system.

Before a final design could be created, it was necessary to decide on a programming language heuristic. This required the implementation of many test systems using functional, visual and object-oriented GUI programming systems.

Ultimately the functional paradigm was rejected essentially because the idea of a user interface did not sit comfortably within a language which maintained referential transparency. Two main solutions have been found to this problem but failed for the following reasons:

- **Unique types:** require the programmer to explicitly declare variables as having only one reference count (i.e. unique). Whilst this maintains referential integrity there is a huge burden on the programmer who must decide which types are unique and then program in an unnatural way to maintain this uniqueness.

- **Monads:** are a way of wrapping program state in a container which hides it from the rest of the (referentially secure) program. The programming of Monads, however, requires an imperative style, which rather removes the argument for using a functional language in the first place.

The visual paradigm was also rejected as the current generation of visual languages merely represent the concepts of current text languages in a visual syntax. This syntax is also expensive in terms of screen real estate and the symbols used to depict concepts are more ambiguous than the equivalent text.

For these reasons, a text-based, object-oriented approach was adopted.

Before any decisions could be made about the specific implementation of the system attributes a way had to be found to integrate language and toolkit. This was, in effect, attempting to solve the problem stated above — namely providing proper interface abstractions within the language.

To determine if any successful abstractions already existed an investigation was undertaken into current professional programming tools, constraint programming and interface specification languages. The abstractions of these systems were rejected for the following reasons:

- **Professional Toolkits:** Toolkits require the programmer to focus on low level presentation aspects, not the semantics of the task the widget is intended to solve. This did not solve the problem of mirroring interface concepts within the language.

- **Constraint Programming:** Whilst constraints have been used to create interfaces for domain specific tasks, generalising constraint solvers to general programming has proved troublesome. It is not that such solvers cannot be built, it is more that they are fiendishly hard to program. If constraint programming were used, it could not scale and would inevitably produce a discrete learning curve.

- **Interface Specification Language:** The use of an abstract interface specification language would allow the interface designer to concentrate solely on the interface semantics. However, should the designer wish to produce an implementation, they would then be required to learn a language and map between the specifcation and the target language. Again, this does not solve the problem of providing sensible abstractions within the language.

Having elminated existing solutions, two novel solutions were proposed to integrate language and interface semantics. To assess their viability, implementations of these solutions were attempted in the next chapter. The issues of language paradigm and integration resolved, it was necessary to suggest optimal implementations for each of the system attributes first discussed at the end of the second chapter. These are summarised as follows:

- **Environment design and execute modes:** A solution was proposed which would present the design and execute modes to the user in a coherent way. This solution was then implemented in the following chapter.

- **Code translation:** Due to its forgiving nature and rapid response, interpreted, or compile-on-demand systems are more suitable to novice programming than standard compliation technology.

- **End user customisation:** To improve end user customisation, the properties of visual components could be given different levels of accessibility, making it possible for the end user to alter aspects of the interface without affecting application execution.

- **Program editors:** Much research has been undertaken into improving editors through the use of outline tools or folding editors. Whilst the pretty printing and syntax colouring of current systems is helpful, much more could be done with the research which already exists.

- **Language structure:** The investigation of the various paradigms, such as functional and visual, would suggest that imperative object-oriented languages are the best suited to graphical interface programming.

- **Language syntax:** Although syntax preference has been shown to be largely subjective, work on programmer psychology has many recommendations as to how syntax might be improved. This has largely been ignored and syntax choices in new languages tend to reflect what has gone before. In summary: syntax could be much improved and the research already exists as to how this can be achieved.

- **Data types:** Different typing strengths are required at different times. Whilst this is provided by variant variables, which then need to be replaced by explicitly typed ones,

a more elegant solution is to take the type checking out of the compiler and place it in the environment, where programmers can alter the strength. The model proposed treats type mismatch as an event, for which different event handlers can be provided depending on the situation.

- **Data structures:** One obvious data structure missing from most systems (most systems having only static arrays) is the list. This is an obvious way to provide a dynamic data structure.

- **Data persistency:** The persistency of HyperCard frees the programmer from worrying about how to perminantly store data conatined in a data structure. Persistent data structures are therefore well suited to novice programming languages.

- **Multimedia data:** Languages for programming multimedia interfaces must have primitive multimedia types. These types, and the operations performed on them, can also be supported directly by advances in chip design, such as Intel's MMX.

- **Error handling:** Error reporting in most modern languages is greatly improved through the use of flexible warning and error settings. These could be further improved for the novice through the use of hypertext links to on-line programming manuals.

At the end of chapter four, the issues of:

- Building interface abstractions into a language
- Removing design and execute modes from interface builders

still required the creation of some viable solution.

6.1.4 New Solutions

Abstracting the Interface

The first attempted solution to the problem of interface abstration involved defining a fundamental widget to which properties could be added, trasnforming it into other types of widget. Sadly this approach failed as no property list could be found which would allow the definition of every widget but would also prevent the specification of non-existent widgets. The second attempt at a solution, however, proved more fruitful.

The problem to be solved was that of representing interface semantics within a programming language. Interface specification languages had acheived this to a degree but the language used to specify the interface was for specification only — no programming was possible. The next logical extension then would be to see if interface semantics mapped directly onto any of the semantics found in programming languages. Modifying the idea from the ACE system of automatically visualising data structures, the hypothesis was that it may be possible to visualise elements of a programming language to create a graphical interface.

Starting with fields and labels, it is easy to see how visualising string variables and string constants would produce these widgets. For radio buttons, there is the equivalent concept of enumerated types, allowing selection of one value from a list. Similarly, for check boxes there is an equivalent in boolean variables.

Besides these simple widget types, the language must also support more complex widgets if it is to be useful in implementing interfaces. This begs the question of how many more widgets is enough? Is there some subset which can be used to implement all other widgets? Fortunately the designers of ACE conducted a large survey of toolkits and found that besides visualising data types, (e.g. string, list of integer, array of boolean) which are already part of the programming language, the following operations must be possible:

- *Select one item from a discrete list*: This is equivalent to the enumerated type.
- *Select a number of items from a list*: This concept is not currently available in programming languages and required the definition of a new concept — the variant enumerated. Using Boolean operators a variable could be specified to hold a list of values taken from a given list. Suppose you want to specify a font style as either plain or any combination of other styles — it could be declared thus: `font = (plain|bold,italic,outline)`
- *Select a single value from a continuous range of values*: Again, this concept was not directly represented in programming languages and required the defintion of a new notation. For example, `a = [5<-|->20]` declares a variable, `a` holding a value between 5 and 20.
- *Select a subrange of values from a continuous range of values*: Once again a new notation was required, which is merely an adaptation of the one presented above. For example `b = [5<-<->->20]` declares a variable, `b` holding a range of values between 5 and 20.

The final type of widget which required a programming language equivalent was the command button. This was found essentially in visualising an object which contained no visual varaibles. If that object contained methods which could be invoked by clicking or selecting then it would appear as a command button. If, however, it did not contain "click" type methods, then it

would be visualised as some form of grouping object — a frame, dialog or window depending on the scoping of the object within the program.

In this way it is possible to produce any interface widget by visualising programming language semantics.

Friendlier environments

The problem remaining from chapter four was in how to present the design and execute behaviours of widgets to the novice programmer. To achieve this, a new metaphor was proposed, based on the users' real world experience.

Using the new metaphor, each interface is equipped with a small screw widget in some corner. Clicking this widget "removes the lid" now showing a small screw widget on each of the individual widgets. The interface functions as normal, until the user clicks on the screw of a particular widget. This widget now enters its design mode, while the rest of the interface continues to execute. The user can alter the unscrewed widget and once complete, can screw the interface back together.

Although essentially a blend of affordances and mode-per-object solutions, it presents the design and execute modes in a way which is consistent with real world experience.

Having produced some novel solutions, it remained to be seen if they actually solve the problems they were intended to.

6.2 Evaluation

6.2.1 Language

The idea of taking some form of structure and visualising it is not new; what is new, however, is producing a programming language which could be visualised. After forming a coherent idea of how the language might appear, a research team (comprised mostly of Dan Olsen's post-graduate students) was contacted to discuss the idea's viability. As this group were working on visualising data structures, they were able to provide experienced opinion on the problems of integrating language and interface. They were able to confirm that my ideas about the language were sound and that, to their knowledge, this had not been attempted before. This provided sufficient motivation to take these ideas forward and implement the

language discussed in the previous chapter. They also reaffirmed the importance of making the tool encouraging to use and to remove kinks from the learning curve. A sample of these conversations can be found in Appendix II.

In creating a language which could be visualised, notations for specifying range, `(a<-|->b)` and the variant enumerated types, `[a|b,c,d]` needed to be designed. These notations were settled on after various samples had been tried on undergraduate programming students. Several notations were abandoned as the students found them ambiguous. (e.g. using different bracket styles for exclusive and inclusive lists). The notations adopted were found to be the most role expressive of those developed. It should be noted that no comprehensive study was undertaken to support this conclusion — an attempt was made by an undergraduate student to undertake such an experiment as their final year project, but the work was never completed.

Having checked the validity of the idea of a visual language, and having designed a notation for that language, it still remained to check that the language could be used to create graphical interfaces.

The creation of widgets through the visualisiation of data structures does not present a problem — the system can match the type of the structure to the type of any widgets available in the toolkit. Menus, radio buttons and check boxes can all be created from the enumerated variable notation — this notation being at least as expressive as the Lean Cusine notation, developed specifically to specify menus [AS89]. Also, by using the range notation, the language can be used to implement the four fundamental widget types identified by the ACE team. By using a "part", these fundamental widgets can be combined together to form most complex widgets; for example a file dialog can be made from a visual part containing a visualised string list (list of files), a visual enumerated variable (for selecting the disk volume), and a visual string (for the file name).

From this implementation work, it would seem that it is possible to build a language incorporating the semantics of the user interface. In the next section, however, we shall see how it compares to current solutions.

6.2.2 Environment

The other major problem which required a novel solution was the integration of design and execute modes. The development of this solution was carried out in conjunction with a company of commercial software developers (see Appendix III) who were interested in providing end user customisation facilities to their software. They felt that the screw solution neatly

solved the problem as it allowed end users to alter the interface without needing to use a different design tool and the inclusion of a single screw did not affect the user's perceived complexity of the interface.

6.3 Have we moved forward?

In the previous section information was presented regarding the development of a new way to build user interface programming systems. Whilst this demonstrates the development of these ideas, can it be said that they are better than their predecessors?

To answer this question, it is necessary to compare the developed ideas against the criteria used for investigation of the previous generation of RAD system.

6.3.1 Goodness and Badness

When surveying the successes and failings of RAD systems in chapter three, the conclusion was that the language should have the following attributes:

- *The language should have no superfluous features inherited from ancestors:*
  The system created was built using existing language technology, as were HyperCard and Visual Basic. Rather than taking a language and adding a graphical environment (as in Visual Basic) or taking a graphical environment and adding a language to it (as in HyperCard), the new language was built by identifying the common elements of graphical environment and language, and using *only* these elements in the language's design. Whilst this does not guarantee no superfluous features, it does provide a way to identify only those concepts required for a language creating GUI programs.

- *The language should contain interface abstractions*:

  When a programmer uses Visual Basic or HyperTalk to create an interface, these languages force the programmer to think of the interface in terms of widget positioning and manipulation. The new language, however, allows the programmer to think at a much higher level and consider the functionality of the interface, rather than its appearance.

  Instead of thinking about presentation details in the initial stages of programming, the programmer can concern themselves with the data structures needed to hold the required data, and the transformations the end user might wish to perform on this data.

Once these aspects have been defined, the programmer can then visualise the relevant concepts and start to consider the graphical aspects of the application.

- *The building environment should integrate smoothly with the underlying language and should not result in redundant concepts*:
  The use of the screw metaphor allows the end user to make the transition to programmer more easily than is possible in current systems. Visual Basic, for example has a completely separate design environment and by attaching the screw to the widget, the problems of choosing either the "button" or the "field" tool do not appear.

  Also, within HyperCard and Visual Basic, the system attempts to protect the programmer from the language for as long as possible. When this protection is insufficient, the programmer must learn the underlying language and hence a kink in the learning curve is produced. By so closely linking the semantics of the interface to the semantics of the language in the new system, it is hoped that the user should become aquainted with the language more easily than in other systems.

  Also, it is not necessary for the programmer to learn an entire language before they can begin to control their application — just as the spreadsheet user needs only to understand the concept of a cell and a value before they can program, the new language requires the programmer to understand the concept of a variable and a part before a graphical application can be created.

6.3.2 Further criteria

Besides the features discussed within the context of UIMS, there were other measures deemed desirable in creating a RAD tool which were discussed in earlier chapters. How well have these concerns been addressed?

Pleasantness

One, rather subjective, criteria for success was the degree of pleasantness, or "positive engagement" exhibited by a system. Is it likely, then, that a system implemented using the ideas presented above would tend to be positively engaging?

By adopting the principles of visualisation, it is certainly possible to put some "instant gratification" back into RAD languages. As has already been noted, one of the benefits of BASIC was that, knowing only the PRINT statement, a novice programmer could create a (very simple) program which produced visible output. The PRINT statement had to be dropped from Visual Basic as there was no default widget or window which could be printed on. By

adding "visualisation" to a language, a novice programmer can create widgets just by using the "visualise" statement.

Visualisation should also encourage exploration of the language as it is possible to visualise every type of variable and each visualisation will produce a different widget[2].

Furthermore, the process of visualisation makes it relatively easy to view the state of every variable, not just those which will appear in the final application. This will allow the programmer to debug their programs more easily than languages without a simple "PRINT" or "visualise" facility.

Of course, the original systems, such as Visual Basic and HyperCard are (initially at least) positively engaging. These systems also benefit from extensive development tools which further increase their plesantness. Because positive engagement is such a subjective concept, using this measure it would be impossible to say that one tool is better than another. Thus the new language does possess some positively engaging features.

Original Criteria

At the end of the first chapter, three guiding principles were defined which described the type of system this work was intended to produce. The success in meeting these principles is detailed below:

1. **Parsimony of concepts:** By producing a system which used only the concepts common to both language and user interface, a system was created which disregarded any superfluous features from ancestral systems.

2. **Supporting a programmer's development:** This was discussed in length with regard to discrete and continuous learning curves. The principles used in creating the prototype language provide a platform from which it is possible to create a tool with a smoother learning curve.

3. **Concept reuse:** The concepts and design principles used in and derived from third generation languages (i.e., data type completeness, universe of discourse, viscosity etc.) were re-interpreted for the RAD paradigm. It was then shown that these principles were misused or ignored in current RAD systems, causing some of the problems discussed in chapter three. These re-interpreted principles were then used to suggest directions for future RAD development.

---

[2]Because *every* variable type can be visualised, the visualisation process will not produce any discouraging, "cannot visualise variable X" type of error messages.

6.4 Further benefits of the new language

6.4.1 Things that went well

Besides addressing the problems discovered in current RAD systems, developing a full RAD system using the suggestions presented in this work would provide the following benefits:

- Because the widget selection is initially automatic for the novice programmer, they are introduced to the most appropriate widget to represent any given concept. Therefore, they are less likely to misuse widgets than someone using current systems. Automatic selection makes it possible to enforce good practice, or a "house style" without forcing the programmer to refer to a list of guidelines.

- Not only are the widget choices improved, but, because the programmer is forced to take a semantic view of interface creation, it is likely that a more coherent interface will be created. This is an example of a "reflexive interaction paradigm" [Thi90] where the language forces the programmer to think at the same level as the user.

- Being able to visualise arbitrary pieces of the program's state not only improves the debugging capability, but makes the language suitable for teaching. In describing requirements for a teaching environment (or "notional machine"), du Boulay [dBOM81] lists the following as an essential attribute: "Methods for viewing selected parts and processes of this notional machine in action." Visualisation ably supports this attribute.

- New widgets can easily be added to the system as the language does not contain any explicit widget references such as "field" or "button". So if, for example, a new widget is added to the system and it becomes the preferred way to display editable text, the system can automatically select this widget and incorporate it into the interface without needing an explicit reference.

6.4.2 At the end of it all

At the outset of this thesis, the author believed that the elegance, terseness and role-expressiveness of functional languages would make them ideal for a novice programming language. Whilst functional languages may possess these attributes, their inability to integrate with graphical user interfaces renders them almost useless as a basis for a RAD system.

Although much insight was gained into the problems of functional languages, their shortcomings (especially those of Clean) delayed the implementation work considerably. It was

hoped that a more complete system could have been developed; this was not possible as it would have required abandoning the original implementation and required a new language and toolkit to be learnt before any new implementation was undertaken.

At the end of this work, what can be presented are ideas on how to design future generations of RAD tools. The ideas presented here have been tested as far as they can. This has been done in two ways:

1. Through consulting commercial developers and student programmers. This has, at least, provided guidance on how to develop these new ideas into implementations which are viable and which would be of use to novice programmers.

2. Through extensive deconstruction of existing systems. Because there is so little work on comparing RAD systems, it was necessary to create some sort of framework for their comparison and attempt to find the causes for the success and shortcomings of current systems. It is inevitable that there are omissions from the comparisons presented here, but at least the language designed meets the criteria derived from other systems

Short of implementing the full system, there is little more evauation work which can be attempted.

6.5 Future work and work in progress

The various systems built as part of this work were intended to check the validity of certain ideas and could in no way be used by a novice programmer. Having tested these ideas, the next step would be to build a system which could be used by novice programmer.

Rather than build an entirely new system based on these principles (a non-trivial implementation task) it is proposed to extend a current programming language with some of the new principles. To that end, the author has already started production of new Java classes which permit programmers to automatically generate interface widgets by "visualising" variables both of standard Java and variant enumerated type. Once completed, this will be incorporated into a lecture course entitled "Device Independent Interface Programming," due to start in February 1998 at Middlesex University.

The ideas behind the "screw" metaphor, however, are not so easily developed as this would require the re-implementation of an entire toolkit. The HyperCard implementation, presented in this work has been continually altered since being originally presented at HCI 95 [Mar95], but can only be extended so far as HyperCard's capabilities will allow.

6.5.1 Related work

One of the most interesting areas of investigation undertaken as part of the work in investigating current end user programming systems were the languages provided on PDAs. Although these are sold as consumer devices, there is a surprisingly rich variety of programming languages available for each platform. These languages, however, are straight ports of desktop languages onto the smaller device, with little regard for the smaller screen or altered widget set of the target device.

It was felt that better languages and, indeed, better interfaces for these devices could be created using abstract widget specifications so that the device could automatically generate interfaces depending on the importance of a given piece of information. These ideas have been presented at the IEE [JM] and a related proposal has been funded by the EPSRC (Grant ref. L70028) allowing the work to be pursued.

6.6 At last

It is hoped that this document has successfully demonstrated that whilst RAD tools are better to use than most alternatives, they have some fundamental, but avoidable, problems. Within this thesis, ideas have been presented which will overcome these problems and which, if adopted by commercial developers, would produce more coherent (and perhaps more pleasant) systems which would better serve the novice programmer.

Appendix I

Language Specification

The work presented here resulted in the construction of a simple programming language, implemented in Clean. A description of this language is presented in the following sections.

BNF specification

The BNF specification for the language implemented in Clean is as follows:

exp ::= *aexp* ['>' *aexp*]
aexp ::= *bexp* ['<' *aexp*]
bexp ::= *cexp* ['=' *bexp*]
cexp ::= *dexp* ['-' *cexp*]
dexp ::= *eexp* ['+' *dexp*]
eexp ::= *fexp* ['*' *eexp*]
fexp ::= *gexp* ['/' *fexp*]
gexp ::= *hexp* ['%' *gexp*]
hexp ::= *iexp* ['|' *hexp*]
iexp ::= *jexp* ['&' *iexp*]
jexp ::= '(' *exp* ')'
| *number*
| *text*
| *boolean*
| *variable*

command ::= *unitcom* [';' *command*]

unitcom ::= *whilecom*

| *ifcom*
| *assign*
| *printcom*
| *codecom*

assign ::= *variable* '<-' *exp*
whilecom ::= 'WHILE' *exp* 'DO' *command* 'ENDWHILE'
ifcom ::= 'IF' *exp* 'THEN' *command* 'ENDIF'
printcom ::= *exp*
codecom ::= 'ON' *variable command* 'ENDON'

prog ::= 'PART' *variable command* 'ENDPART'

Variables and typing

The language is weakly typed with an automatic coercion scheme, where all types are ultimately converted to a string (similar to that employed in SmallTalk). This scheme was chosen simply because programs would not crash due to type mis-match errors. To support the ideas of providing a language which could exhibit different strengths of type matching, the coercion routines were provided in a separate module and were re-written to provide strict typing should it be required.

Declaration of varaiables was implicit, a type being allocated to the variable depending on the value assigned to it. Variable scope follows normal lexical rules.

The store for the language was organised to retain everything as an object. Objects could be simple variables (having an attached value) or could have methods and attributes also stored. These were then visualised by pattern matching the representation in store. The store also permitted a message passing hierarchy through enclosing parts.

Driving the language

The language "environment" will read an ASCII text source code file (prepared in some external editor). If the program loads successfully, it is displayed on a text window (editing of the program from this window is not possible).

The only other option is to execute the program. This will either be successful or result

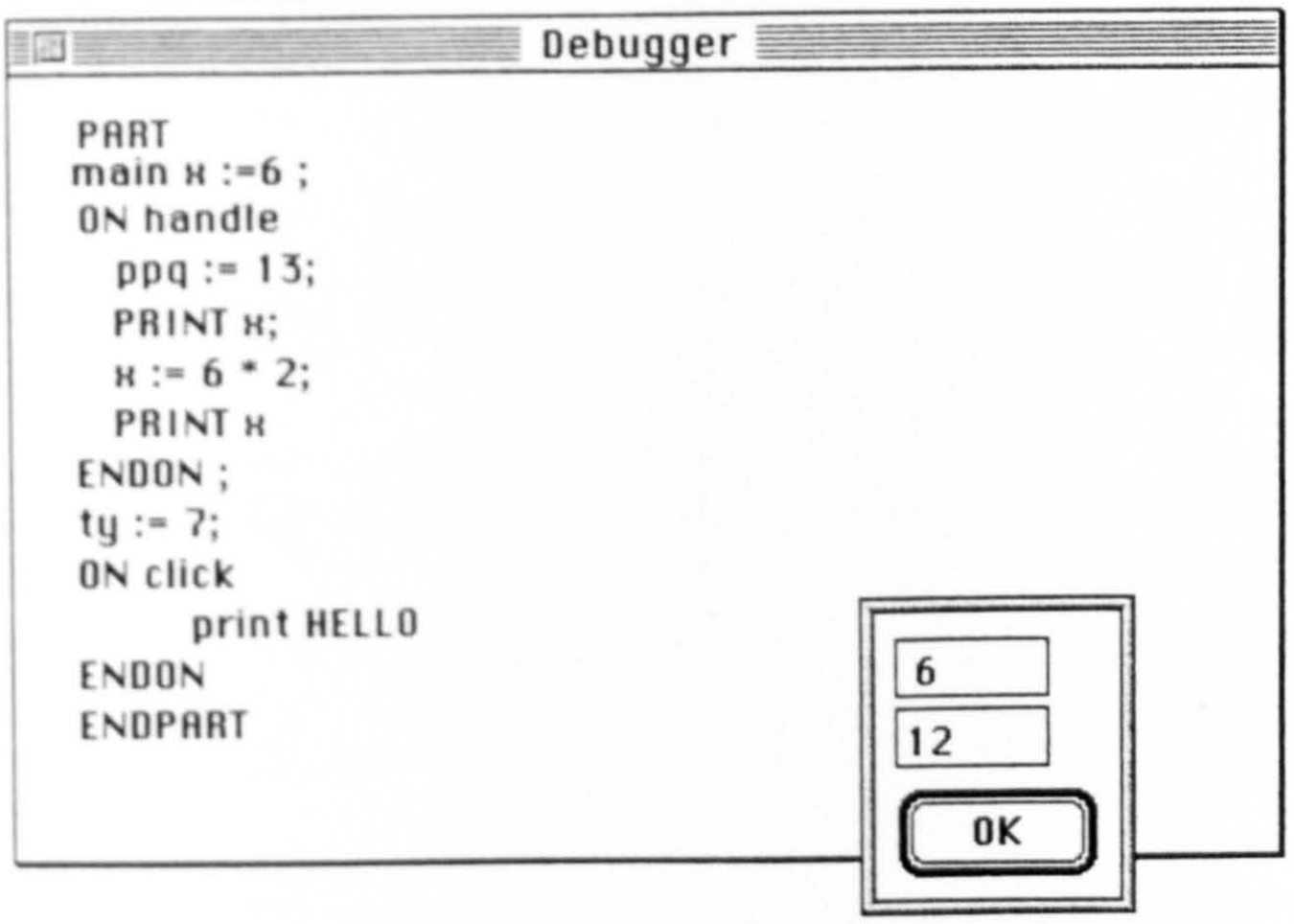


Figure 6.1: Executing the language implemented in Clean

in an 'error.' Figure 6.1 is a screen shot of a simple program being loaded and executed; the program is displayed in the "Debugger" window and the dialog box is displayed when a "handle" event is generated.

Shortcomings

As this language was being written, the implementation language, Clean, went through a major revision ($0.8 \rightarrow 1.0$) which required that the entire source code be re-written. Before undertaking this process, the language was able to automatically visualise expressions, choosing different widgets for different variable types and different string values. This is the version of the language for which the BNF specification is provided.

The current version of the language, now implemented in Clean 1.2, should also be able to visualise code "Parts" but due to the uniquely typed I/O system, this has proven almost impossible to implement. Because of this, the visualisation of code parts was achieved in Java using pre-assembled objects.

As was stated in chapter six, one could easily reimplement the entire language in Java to overcome this limitation of Clean.

Appendix II - Letter from Neural Innovation

Neural Innovation

Neural Innovation Ltd.
The Alpha Centre
Innovation Park
Stirling, Scotland
FK9 4NF

Tel. 01786 446920
Fax 01786 446921
Email mail@neural.co.uk
URL http://www.neural.co.uk

To whom it may concern.

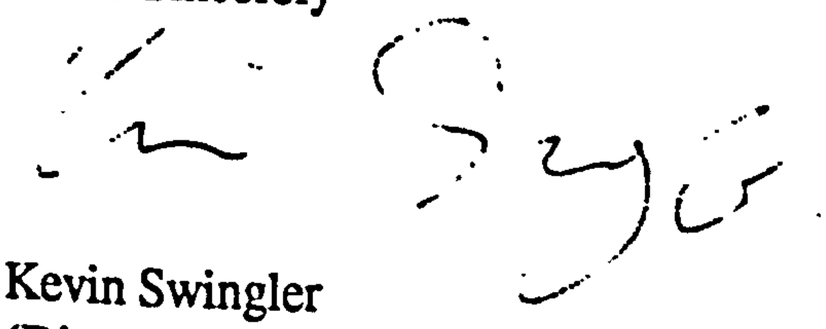
March 4, 1998

This letter is produced in response to a request for information concerning our studies of the work of Gary Marsden. Our company operates as a software development house for neural network based data analysis tools. Throughout the period of his PhD, Gary Marsden discussed the ideas in his work with respect to our visual interface design. In a number of cases, he created prototype examples of potential user interfaces using his methodology.

We found his work to be insightful and think that his thesis contains a number of ideas which would improve the design and usability of GUI software. We were particularly impressed with the concept of using screws as a visual metaphor to hide or expose underlying blocks of a GUI system. We think his methodology could provide a quick and understandable method of putting together GUI systems.

To conclude, having seen Gary's work, it is my professional opinion that his ideas are of direct relevance to commercial Graphical User Interface design.

Yours Sincerely



Kevin Swingler
(Director)

Appendix III - Email from Viewsoft

Sample Email conversations between myself and a professional developer at Viewsoft .

Please note that the included '>' text is my side of the conversation.

>I have just read an article in BYTE (vol.20, no.4) reviewing your Utah
>product. Unfortunately for me, it sounds identical to the software I
>have been working on in my PhD. I am a final year PhD student at the
>Department of Computing Science, Stirling University, Scotland. I have
>been developing a language for programming direct manipulation
>interfaces which can automatically allocate widgets to visualise parts
>of a program, based on the semantics of that part of program.

I'll send some literature that tells about the product in the mail.
Also, you can download our white paper and other interesting
items from

ftp.itsnet.com

in the viewsoft directory

I would be glad to correspond with respect to our work after
you have read the ftp literature.

-Kelly

>I have read the ftp papers now and found that starting from the same
>point, our work has diverged greatly (much to my relief!!).

Not too surprising. There are a million ways to take this stuff. I hope you
get to graduate still :-)

> I agree with
>everything you say in the white paper: that it is senseless to create a
>program variable and a widget to view that variable, when the only
>purpose the widget servers is to view the value of the variable.

I'm not sure I understood this quite right. Perhaps it was poorly worded,
but creating a variable in your program and having views for it on the
screen is in fact the whole idea of our product, so calling it senseless is
not what I am trying to say in the white paper.

If what you are saying is that if that is the case, why not just make the
whole language visual, then you are indeed doing something very different,
and far far more challenging. We believe that some things are expressed more
succinctly and accurately using a textual language. Having used visual
languages such as Borland's Object Vision, and NeXT Step (only partially
visual), Serius (now Novell's Appware) and CASE tools, I find them at least
as complex as textual languages. My personal belief is that a purely visual
language is difficult, because it won't be powerful enough for programmers,
and it won't be simple enough for typical end users. So you are left with a
few souls in the Netherlands in between. Obviously, this is where visual
basic currently lives, so it isn't empty ground, but I ramble... :-)

>The
>route I have taken with these ideas is to design an end user programming
>language, based on the idea of visualising program components: declare a
>variable and then visualise it. Declare a variable of an enumerated type
>(say [Plain | [Bold , Italic , Outline]]), visualise it and have the
>system pick the most appropriate widget.

We initially have the system pick the most appropriate widget, but have mostly abandoned the idea that the computer could ever create a "good" user interface without a lot of human assistance.

>There are all sorts of benefits
>in end user programming using this principle. I am currently finishing a
>paper discussing how the semantics of buttons and dialogs can be thought
>of in a similar way of visualising program fragments.

This sounds similar to Olsen's work, only working from the other end of the same problem.

>I am fairly familiar with Olsen's work which, like yours, seems aimed at
>the professional programmer.

Definitely, we are primarily aimed at the professional programmer, for now.

>Your product sounds very
>interesting and I shall try to talk the department here into buying it
>(it would help a lot with the implementation of my language).

That would be nice. :-)

>I don't
>know if you are at all interested in what I am doing, but will happily
>send you a copy of my next paper.

Very interested. We intend on moving into the end user arena, as soon as our base technology is strong enough.

-Kelly

After sending them a draft of my ideas

>Sorry, I didn't explain myself properly. I too think that it would be
>impossible to build a completely visual system. Packages like Authorware
>merely swap text for visual syntax (not as explicit, or compact as text)
>which just plain infuriates people. Other systems, such as by-inference
>things merely abandon people when the programmer's needs become greater
>than those the system can provide.

This shows a lot of insight that is often missing in academic circles. Its amazing the difference a couple of years of getting beat up by customers will make :-)

>What I tried to say before was that if you are going to create a visual
>programming system, it becomes apparent that it would be convenient to
>merge the concept of variable and widget. As you said in the white

>paper, the system can take care of keeping the widget view of a variable
>up to date, without the programmer having to explicitly write code to do
>this.

The idea of merging the variable and the widget is interesting, but one of the key benefits of not merging them is the ability to have multiple views on the same data. User interfaces are replete with examples of several different views on the same data. If you merged the widget with the variable, it seems that multiple views would be lost.

>Sorry, no revolutionary ideas on this front :-(

I didn't really expect any :-)

>What it does provide in
>an end user system is confidence booster and donkey-and-carrot effect.

Nice. The thing I like about PhD types is that they often bring up these psychological angles on this stuff. Very good piece of work. seriously.

>Knowing only about variables, a user can start to create a visual
>interface. OK, so it's not going to be a very good interface, but at
>least they can see an immediate reward for their work and learn
>something about the use of what widget is used for what purpose.

I thought immediate gratification was an American thing :-)

>I teach
>programming here, and it is very frustrating for novice programmers to
>learn lots of Pascal syntax before then can even print out a variable.
>They compare this to the programs they see running under windows and
>wonder how they can ever write programs which look like that.

There is currently a very, very steep learning curve to getting up to speed in both a new programming language (often C++, sometimes visual basic) and getting up to speed on Window's version of event based programming.

>I wonder
>if I had started out on Pascal instead of BASIC, would I have stuck with
>programming. Sorry, slipped into righteous rant mode there. :-)

Probably. But I see learning programming as a little bit different problem than writing good programs. Granted, they are related, but you probably shouldn't use the same tools for both, unless you have a really, really good tool. BTW, I don't think such a tool exists yet, including ours and probably yours. I think we are both moving in the right direction, but it takes many many man years to get that good and we've only spent about 20 so far.

Appendix IV - Published Papers

Designing and Interface Programming Language for the End User

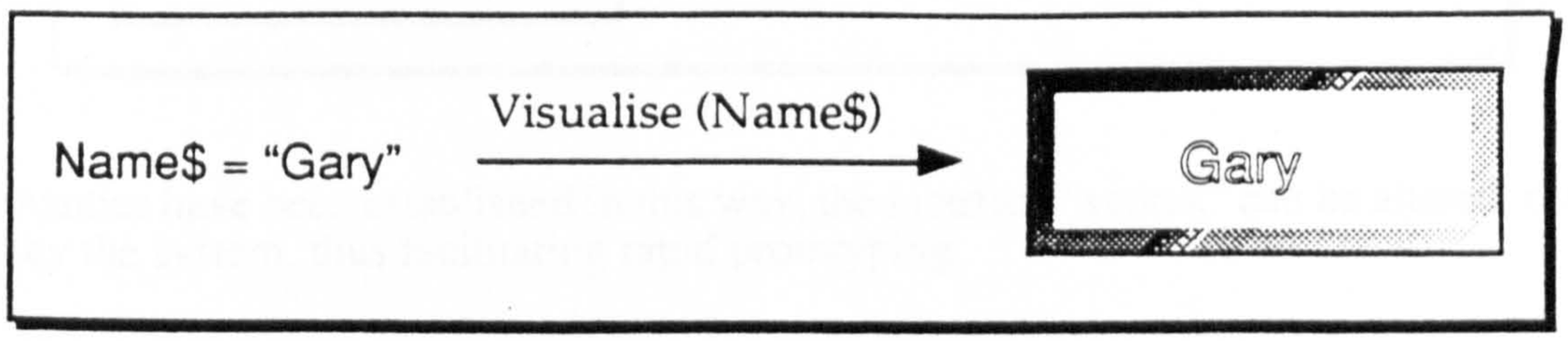
Gary Marsden & Harold Thimbleby
University of Stirling

Current Problem

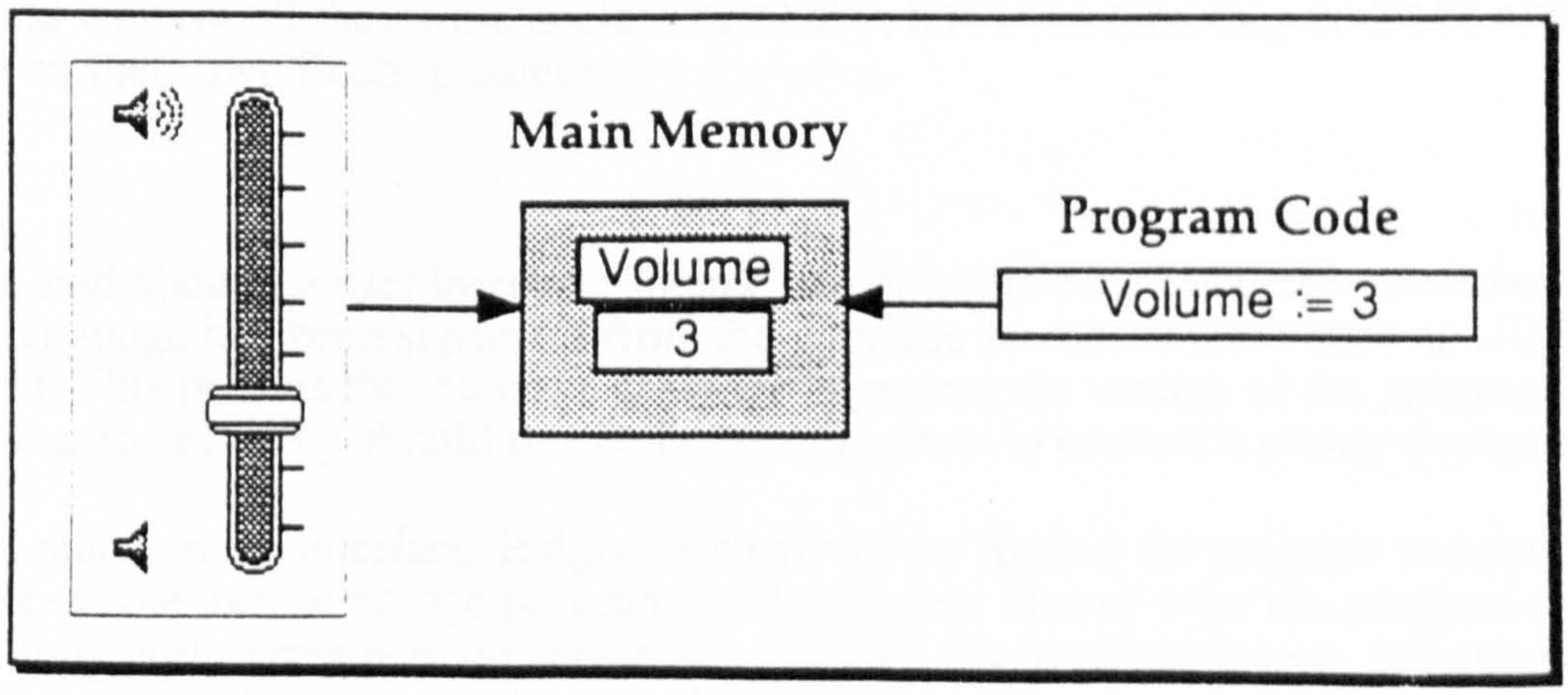
HyperCard and Visual Basic have shown that small languages for programming GUIs are very popular. With powerful, graphically based computer systems becoming widely available, novice users have no simple way to program them. Most programming systems are built on a traditional third generation language, such as C or Pascal, and have either interface builders or libraries cobbled on top. This results in (1) A *huge step in the* learning curve as programmers abandon the interface development environment to learn the underlying language; *(2) Languages designed to cope with* character and file I/O are being stretched to cope with I/O demands they can never hope to meet. *What is needed is a new I/O paradigm,* allowing languages to easily create direct manipulation interfaces.

Grounding for Solution

The state of an executing program can be determined from the values contained in the program's variables. As the purpose of many GUI widgets is to reflect the state of a program, it would seem sensible to provide an easy way of reflecting the values of variables directly on the screen. Therefore, in the system currently being implemented as part of this work, a screen widget is not explicitly created. It is created implicitly by asking the system to "Visualise" a variable. The type information for a variable is used to provide a default widget, which can be altered once the programmer learns more about the system. (For instance, a HyperCard locked field, or Visual Basic label would be used to display a string value).



Direct manipulation interfaces (as their name implies) not only reflect a program's state, but allow it to be altered using widgets to display and edit a program's variables. Consider the "Volume" setting in the Macintosh Sound control panel. It is a program value edited by a slider widget, but can also be changed by the program.



The are many advantages to this approach.

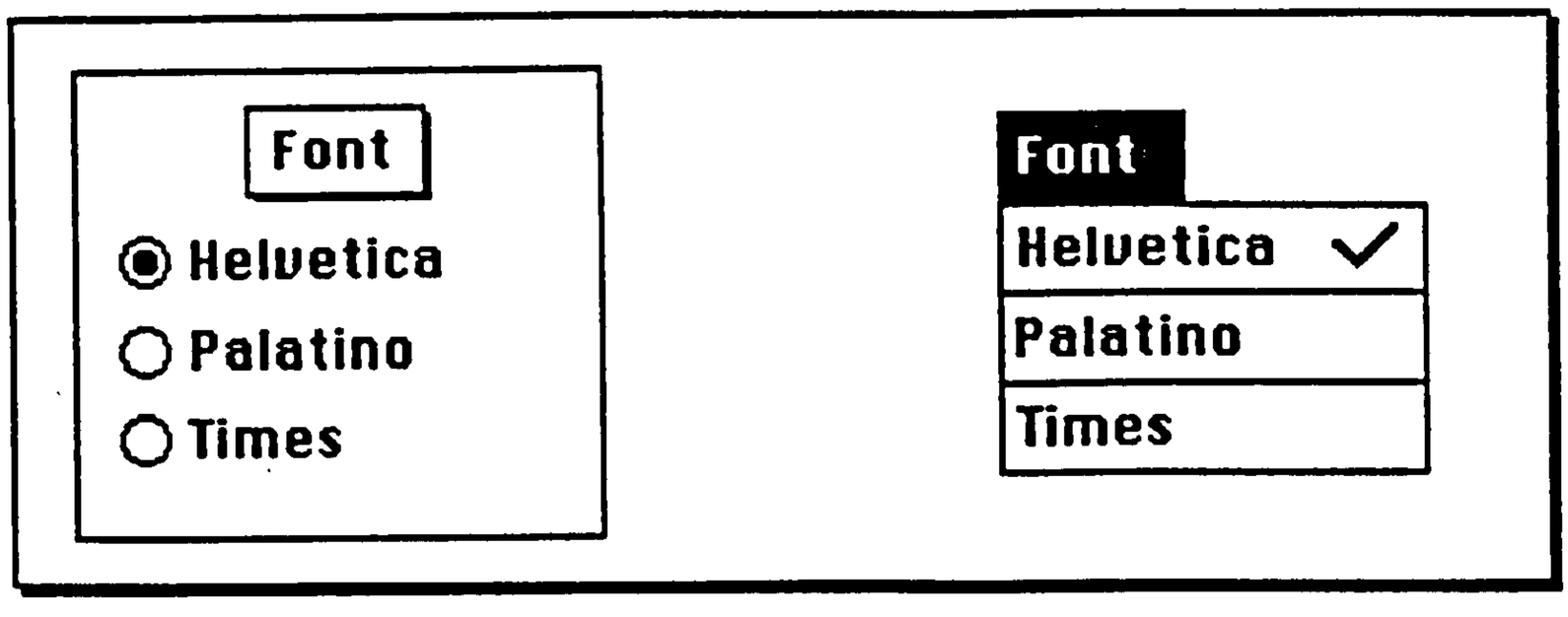
1) Programmer focuses on semantics of the application, not its "Cute Graphics."
2) The user interface is guaranteed to be an exact representation of the program's state (the programmer no longer has to explicitly connect call-backs to variables).
3) As the system is providing defaults, the user can learn quickly, and tailor the application as expertise increases.

Complex Widgets

The paradigm of visualising simple variables can be extended to encompass more complex structures. Two of the most common forms of widget which must be represented are: (1) Selecting one value from a group; (2) Selecting many values from a group. To illustrate these differences, consider a font menu containing font name and font styles. Obviously characters can only be of one font (eg. Times), but can have many attributes (Bold, Underline). The system at present allows the declaration of variables in a tuple, using Boolean connectives to distinguish mutually exclusive elements. This is illustrated below.

```
Font := [ Helvetica | Palatino | Times ]
Style := [ Plain |  [ Bold , Italic , Underline ]  ]
```

As an example, the font menu may be visualised as either:



Once the semantics have been established in this way, the interface "syntax" can be altered easily and consistently by the system, thus facilitating rapid prototyping.

Utilising Scope

Another attribute of variables (and sub programs, such as procedures) which may be used in choosing a representation, is their scope. First consider the notion of a visible sub-program. This would be represented as a form, or window within an application. As heap space is allocated and de-allocated for sub-programs, then any widget created within their scope is automatically created and destroyed. The menus in the previous example, if declared in a local scope, could only exist as buttons or a pop-up menu on the associated window. If they were declared globally, however, then they could be either application menus or exist on their own floating palette.

Conclusions

Much has been said about the user interface "being" the programming language. Until now, the programming language had been separated from the interface by mechanisms such as UIMS and call-back procedures. This permits the interface designer to present the version of the program's semantics they wish the user to see. Why should this be done, other than to conceal a poorly designed program?

The imposition made on the interface designer's creativity by forcing the program semantics to be directly represented, means the user or novice programmer has a clear idea of what the program is doing. Indeed, as there seems to be little agreement on semantic guidelines for interface design, this approach is certainly no worse than any other. It does have the benefit, however, of drawing on the considerable research conducted in programming language semantics.

PAGE NUMBERING AS ORIGINAL

Overcoming Design and Execute Modes in User Interface Design Environments

Gary Marsden
Department of Computing Science
University of Stirling, Stirling,FK9 4LA
email: gaz@cs.stir.ac.uk

1 Introduction

1.1 The Problem

Direct manipulation interfaces exploit the user's knowledge of real world objects by using analogous, or metaphorical, on-screen representations of these objects. Provided the analogy is strong, the user can interact with the software without any explicit training. Problems occur, however, when the metaphor falls apart and the screen object behaves differently than the user had anticipated. Classic examples, such as the Macintosh trash can ejecting a disk, can seriously undermine a user's faith in what they have learnt to date about a system.

•

This is especially true when a user wants to start customising and writing their own interfaces. Novice programmers who use an interface builder for the first time encounter a unique problem. Rather than the expected response of highlighting a box, clicking a check button in the interface builder environment causes a check box property dialog to appear. The analogy between real world and on screen objects can no longer be maintained. If the programmer is to successfully alter and create interfaces, they must learn that widgets have an 'execution' behaviour (with which they are already familiar) and a completely new 'design' behaviour. It is this new design behaviour they must master in order to control the execute behaviour. Of course, this is really just another manifestation of a 'mode' problem. The uniqueness of this application, however, has produced a variety of solutions, none of which could be termed optimal.

1.2 The Ideal Solution

Solutions to this problem must either complete remove this modality from the interface, or find a consistent metaphor for the design/execute behaviour. It should also be remembered that the interface builder is being designed to aid the transfer of a GUI user to a GUI programmer. This is an entirely different proposition to designing an interface builder for a programmer wishing to write GUI programs. Any solution must permit a the user /programmer a gradual transfer from simple interface alterations to full interface programming.

2. Current Solutions

2.1 Modal

One way of tackling the mode problem is not to tackle it at all. After all, every programming language has a design and execute phase. Visual Basic [3], therefore, has very clearly defined design and execute modes. The programmer builds the interface, writes the code, then enters the Run command (as with more traditional versions of Basic). Although aimed at the novice programmer, the different modes mean that Visual Basic can only be used by those willing to learn a full programming language. HyperCard [4], Apple's alternative to Visual Basic, also exhibits application wide design and execute modes, but provides a more gradual introduction to GUI programming.

Within HyperCard, programmers choose from a selection of tools which allow them to alter the mode of interface interaction. Clicking a button with the browse tool will causes the button to respond with normal, execute behaviour. Clicking the button with the button tool, however, will allow the appearance and behaviour of the button to be altered. They can migrate smoothly from an interface user, through various levels of tool complexity (five in all), until they can confidently program the interface.

The main problem with both these solutions, however, is that no attempt has been made to remove the mode from the interaction: novice programmers will still see inconsistent widget behaviour. This problem is exacerbated by systems which fail to adequately convey which tool, or mode, is currently active. Studies with interface builders report that using some systems, even expert programmers can have difficulty determining which mode is active [1].

An alternative to having system wide mode, is to introduce modality on a per-object basis. This would seem to be more in keeping with the real world, where it is perfectly feasible to paint (deign) a wall and listen (execute) to a radio. It also has the benefit of removing the tedious mode changes encountered when developing an interface in a system which uses a global mode change.

2.2 Affordance

An alternative to mode based interaction, is the provision of affordances on objects. This has been used successfully in applications, such as window managers, where there is a tightly constrained set of customisations a user can make. By adding close, zoom and resize boxes to a window, the user can perform modeless customisations on that window. The affordance solution, however, cannot be easily applied to a programming environment. Adding customising affordances to each widget would clutter the interface and hopelessly confuse any end user. (There is also a problem in customising the affordances - for example, how do you change the size of a close box? Provide a meta-affordance?)

2.3 Buttons

In [2] Smith proposes a solution based on buttons which apply customisations to widgets. An interface programmer / user can drop customising buttons on objects, which change that object's behaviour. The customising buttons, argues Smith, can be thought of tools, and thus the solution is very like the user's real world experience. Certainly, such a solution does not introduce any mode, as the tools' are part of the executing interface. By providing tools of varying complexity, users can easily make the transition from simple to complex interface alterations. The idea of a button representing a tool, however, does seem to unduly stretch the programmer's experience of the real world. This does not bother Smith who seems to equate an intuitive solution to one in which objects have no modes.

3. Novel Solution

If I wish to repair or alter an object in the real world (say the computer on which I am currently typing), I immediately examine the casing in an effort to find a screw, or similar fastener, which allows me to take the object apart. By removing all the external fastenings, I have access to the mother board and all the sub assemblies inside the machine. These, in turn, have more fastenings which I can remove until the component I am interested in can be found. This same approach can be applied to editing and executing a direct manipulation interface.

Imagine an interface which is in 'execute' mode, with a single screw affordance visible on its surface. Clicking the pointer on the screw widget causes the screw to revolve and pop out of it's hole. The surface of the interface slides back to reveal screw (such as menus and frames) on individual widgets and sub assemblies[1] and also a tray of available new parts. Clicking the screw of an individual widget reveals a property sheet for that widget, allowing the user to change its various properties. All other widgets on the interface will continue to retain their execute behaviour. Clicking on a unscrewed widget allows its position to be altered by dragging it to a new location. Once alterations to a widget are complete, it can be screwed back in to the interface (see Figure 1 for an example interaction). This screw metaphor can be extended in several interesting ways:

- By providing a range of fastenings from a knurled nut through to a torx screw, a variety of customisation levels (from changing colour to full programming) can be provided. Not only does this allow users to gradually learn about interface programming, but gives feedback as to the complexity of the customisation they are about to undertake. This is not the case with [2].

- The interface can be set up so that screwing in the original, top surface, screw causes all other screws to be automatically screwed in.

- Rather than providing a property sheet, it might be possible to provide special customisation widgets (such as potentiometers) as found inside real electronic equipment.

4. Conclusion

The interface metaphor discussed is essentially a blending of a mode-per-object and affordance solutions. By providing a single, generic affordance on each widget, potential cluttering is eliminated. The affordance can then control the mode of the widget, denoted by the screw state (in or out). This mixture provides an environment which is consistent with user's real world experience.

N.B. In the development of this idea the author was given various 'useful' suggestions as to how the realism of the metaphor might be improved. These included: have springs fly out to be irretrievable lost whenever the lid is removed; having screws randomly disappear; leaving several screws behind when the lid is closed. There are no plans to incorporate these features into the finished system!

References

[1] S. Houde & R. Sellman: In Search of Design Principles for Programming Environments, *Proceedings* ACM CHI'95, pp. 424-429.

[2] R. Smith, D. Ungar & B. Chang: The Use-Mention Perspective on Programming the Interface, *Languages for Developing User Interfaces*, pp.79-89

[3] Microsoft: Visual Basic Programmer's Guide

[4] Apple Computer: HyperCard User's Guide

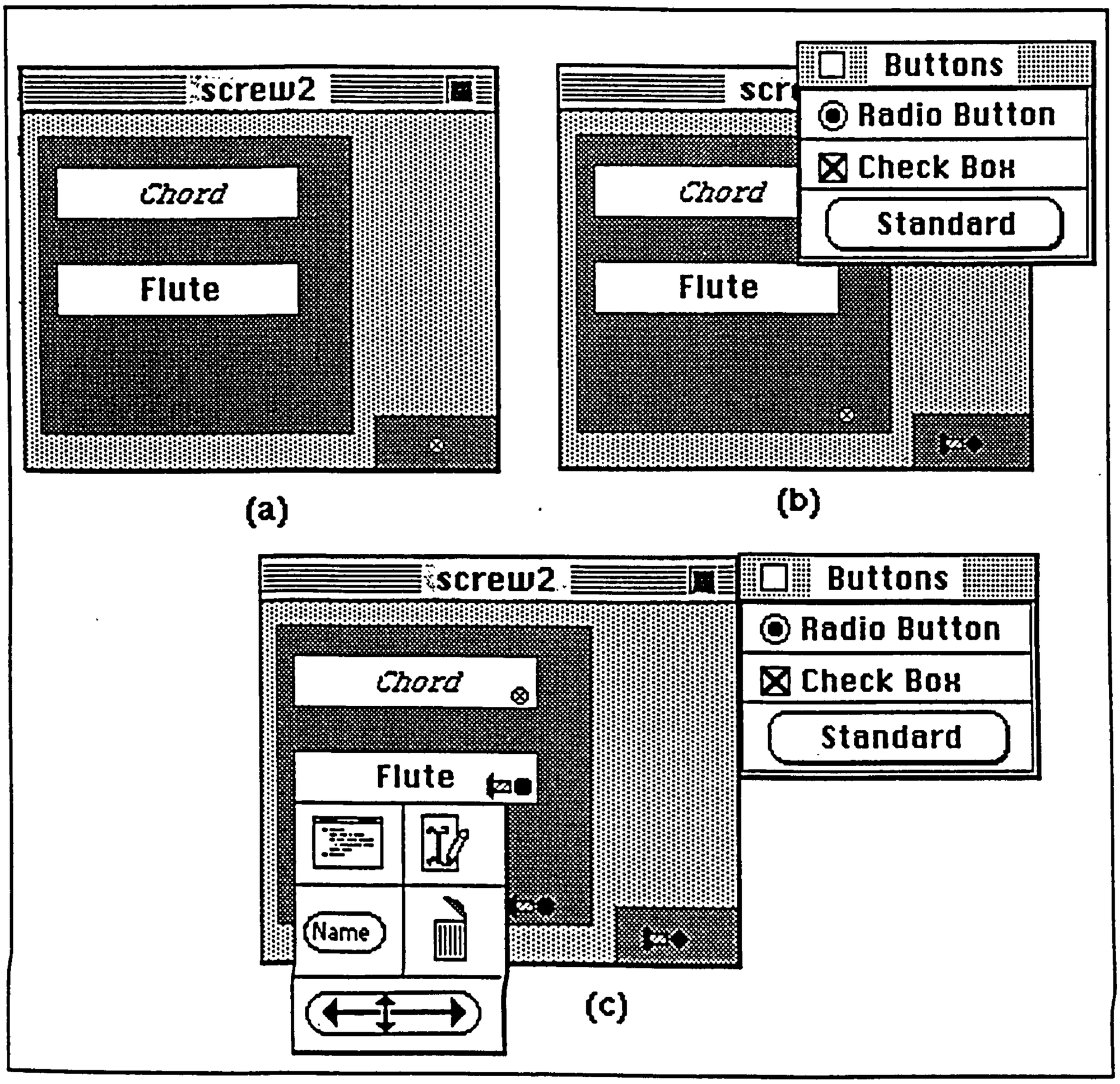


Figure 1

(a) Initally only one screw is visible. **(b)** Clicking the screw reveals other screws and part trays. **(c)** Unscrewing an the 'Flute' screw causes a property altering palette to appear for the 'Flute' button. The 'Chord' button still works as before.

Resource Enhancement - Overcoming Limitations of Software Packages

Gary Marsden, School of Computing Science, Middlesex University, Bounds Green Road, London, N11 2NQ, England *email:* g.marsden@mdx.ac.uk

Karen Chan, Department of Computing and Mathematics, University of Stirling, Stirling, FK9 4LA, Scotland *email*: kch@cs.stir.ac.uk

Abstract:: Packages such as AuthorWare™, HyperCard™ and Visual Basic™ are designed to support the rapid creation of highly graphical, direct manipulation applications, such as might be used in the creation of tuitional software. The nature of this software is so diverse, however, that often a single package cannot provide all the facilities required. This paper discusses how to overcome limitations in individual packages and how to 'glue' several packages together to provide the required functionality. The methods discussed have been used in providing software for teaching the mathematics degree at the University of Stirling.

Introduction

There are several software packages available which have been used in the creation of Computer Aided Learning (CAL) software. These packages allow the programmer to create "displays" of information, screens full of multimedia resources and material. If the programmer wishes to make their software a little more interactive, these authoring packages often lack the adaptability to provide the necessary computational power. Rather than re-implemeting the entire package (both time consuming and possibly beyond the capabilities of the programmer), this paper examines ways in which it may be possible to overcome the limitations of these authoring systems.

One possible way of overcoming these limitations is to extend the functionality of the package with specialist components. Many packages allow new pieces of code to be "plugged in" to extend the original functionality. These components are typically written in a high level language such as C or Pascal and allow the developer to implement the required features. Fortunately standard forms of these plug-ins exist (VBX, DLL, XCMD) to standardise external components allowing the same code to be used with a number of software packages. As an example of this type of enhancement, we shall look at the HyperCard XCMD and the viability of reusing a HyperCard XCMD within AuthorWare.

Another common solution is combining resources from separate software packages to overcome limitations in any one package. Using high level communication protocols built into the operating systems.

Enhancement By Adding Custom Software Components

Exchanging External Resources Between HyperCard and AuthorWare for Macintosh

The Background

MacTutor [O'Connor 1996], developed by a group of Mathematicians from St Andrew University, is a mathematics tutoring package written in HyperCard. At Stirling University this package is used frequently in the undergraduate mathematics teaching. MacTutor is primarily written for mathematics teaching and consequently, there are limited resources in the package suitable for teaching statistics. One undergraduate course, however, required the development of a drawing package to produce some standard probability distributions, and to fit probability models to a set of real data. The drawing package was required to illustrate, in the continuous case, that probability is the area bounded by a curve and two specific points, selected by the students. One crucial requirement for the program was that it be able to display these graphs in real time, reflecting changes as the parameter, or parameters, of the probability function are altered. Unfortunately, neither of the two courseware authoring packages favoured at the university, HyperCard [2] and AuthorWare [Macromedia 1997], have the ability to meet these specifications. It therefore became necessary to enhance one of these with external resources – in the end, AuthorWare was chosen.

Choice of Authoring Tool

There are a number of reasons why the package was not developed in HyperCard. Firstly, for this particular application, the HyperCard "card" metaphor is inappropriate; each screen requires a new background. Integrating colour into stacks is also a problem with HyperCard. Version 2.1 of HyperCard, in which this application was initially written, provides no support for colour. Although colour is supported in HyperCard 2.2 & 2.3, the implementation is both awkward to use and slow in execution.

Other factors against using HyperCard include poor animation support and a contrived method of displaying colour pictures. AuthorWare seems to support these requirements, reducing the effort required to produce this type of software, especially for those who have minimal programming experience. However, this is not to say that AuthorWare is perfect. Although its interface is easier to use, it is primarily a multimedia authoring package and hence lacks any serious computational ability.

XCMD's and XFCN's

XCMD's and XFCN's (eXternal CoMmanDs and eXternal FunCtioNs respectively) [Apple 1993] are code resources written in a high level language which are compiled into a stand alone module. XCMD's can be written in almost any programming language although C and Pascal are commonly used. They can be linked to any Macintosh application complying with the 'X' interface protocols. Typically, the command or function will appear as a new keyword or function call from within the host application. So far as the developer is concerned, the calls to external resources are identical to those made to the standard application resources.

Within MacTutor, there are many useful XCMD's and XFCN's, including graphical display tools, which would be useful in developing a drawing package. For example: "graphit" - which draws a function's graph and "integrate" - which integrates a function and draws the area on the screen. Although it was decided to develop the course software in AuthorWare, an investigation was made into the reuse of the MacTutor / HyperCard XCMD's in the AuthorWare environment. Fortunately, it was found that AuthorWare does support most of the HyperCard 2.0 XCMD interface (see AuthorWare User Manual for further details).

Implementation

External resources are reached through the "Load Function..." option in AuthorWare's "Data" menu. As HyperCard is based on a text language (HyperTalk) and AuthorWare has an iconic language, it was necessary to create a "calculation icon" within the AuthorWare program to communicate with the XCMD. The calculation icon was set up to contain variables representing input values to, and return values from, the XCMD.

The "graphit" XCMD imported into AuthorWare requires the setting of maximum and minimum values on the X and Y axis, and also the specification of the function to be plotted. (Students can alter the range of x- and y-values to be plotted using *Zoom In* and *Zoom Out* buttons on the screen). An interesting feature of "graphit" is that it must be used in conjunction with another XCMD, "axes" which is responsible for plotting the axes. Both these XCMDs behaved as expected, despite the fact that they were being used outside their original environment.

Code Resources in Windows

As AuthorWare is available both on the Macintosh and PC/ Windows, applications developed on the Macintosh can be converted for use on the PC with minor alterations. (Note that AuthorWare 2.0 does not support conversions from Windows to Macintosh.) When a file is converted to Windows, XCMD's are not convertible simply because the are stored as machine code, specific to the Macintosh hardware. It should, however, be possible to transfer the source C / Pascal code of the XCMD into a Windows compiler, and recompile the code into a Dynamic Link Library (DLL). DLL's are linked into a program as it is running. This has many benefits in that: disk space can be saved by placing duplicate code from separate applications into one DLL; facilities may be added to an

application which the original authors had not considered; memory can be saved by removing unused DLL's from an application.

A further way of sharing code resources in Windows has resulted from the popularity of Visual Basic, which was designed to allow the inclusion of third party widgets into its environment. VBX's (Visual Basic eXtensions) [Feldman 1993] are separate code components, which enable the creation of customised software by providing specialist interface elements and code resources. These range from Neural Network software to Telephone exchange controls [Udell 1994]. The main difference between these and DLL's is that VBX's require the run time support of the Visual Basic environment. Other packages wishing to use these controls (eg. Visual C++), must in some way emulate this environment, which can be a daunting task. For this reason, only a few applications have succeeded in utilising VBX controls.

It should also be noted that the programming interface to VBX controls is not as clear as might be hoped for. Notional, VBX's have both properties and functions, one for setting attributes of the VBX ('colour', 'size' etc.) and one for making the VBX carry out a task ('draw graph', 'connect to server' etc.) In the authors' experience, however, the programming interface to VBX's is not so well defined. In the standard graph control which ships with Visual Basic 3.0, there is no function 'draw graph' when new data has been entered - rather the property 'draw mode' must be set to 4. In other words, the 'draw graph' function is provided as a side effect of setting the value of a variable!

With the release of Visual Basic 4.0 a new type of VBX control has emerged. This new OCX is a 32 bit only control, relying on the OLE protocol which is built into the Windows operating system, and most windows applications. Undoubtedly this standard will grow in importance over the next few years, but VBX will remain with us due to the legacy of 16 bit systems.

General Comments

In the wake of the software crisis, many new methods were proposed to improve the quality of software and the ease with which it is produced. Object orientation was one such proposal, claiming, amongst other benefits, that programs could be decomposed into "objects" which could be shared and re-used by different applications. Both VBX's and XCMD's are, perhaps, the simplest form of objects, they go some way to fulfilling the potential of object oriented systems.

Enhancement By Combining Elements From Generic Packages

Instead of relying on custom functions, current operating systems allow applications to overcome their limitations by accessing facilities provided by other applications. For example, an email program may not implement any spelling checks, but ask a word processor to perform this task on its behalf.

The ideal situation, therefore, is for the operating system to provide a directory of services available from the currently installed applications. Programmers can then access these services from within whatever system they have chosen as a development environment. Whilst this is indeed a grand concept, its current implementation is hampered by the following factors:

- To have all services available, the server applications must be concurrently executing. At the time of writing, the vast majority of systems used for executing this software (Windows, Macintosh System 7) do not support full pre-emptive multitasking. This makes for unreliable and slow execution of concurrent applications.

• In order to provide a directory of services, application developers must explicitly provide support for the operating system's high level communication protocol. Often packages do not support high level communication as the developers decide that the perceived benefits do not justify the associated implementation effort.

• There is a current trend to "Fatware" [Udell 1993] applications: single, huge applications which try to anticipate, and provide for, every need of its intended user. Not only do these applications inevitably fail to do everything every user wants them to, but they make such intensive use of hardware resources that it is often impossible to run more than one application at a time.

• Unless sold as a suite (such as Microsoft Office) application developers cannot rely on the required applications being available to provide funtionality not built into their application.

Clearly, trying any form of high level communication between applications will be a tricky, uncertain artl until the legacy of single tasking operating systems and monolithic applications has been left behind. It is interesting to note, however, that large applications favour application developers (more features look impressive in advertising; increased value software; new training courses required for more complex software) and hardware manufacturers (faster, more powerful machines required to run resource intensive software). This may still prove to be a factor in preserving the status quo. Hope for the future is provided by systems such as OpenDoc [Apple 1997] and Java [Sun 1997] which promotes the development of small applications, appletes, dedicated to performing specific tasks.

In a strange compromise, Microsoft has committed itself to OLE (Object Linking and Embedding). This is a protocol which allows the embedding of objects from the document of one application into the document of another, the classic example being an Excel graph in a Word document. This may prove of great benefit in the future, but currently the technology has not been widely adopted. This could be primarily due to hardware load as all the applications involved in the embedding must be executing concurrently.2 A direct comparison of OLE and OpenDoc can be found at [IBM 1997]

High Level Communication on the Macintosh Using System 7

Using OSA

HyperCard for the Macintosh already includes a high level scripting language, in the form of HyperTalk. It is hardly surprising, then, that Apple first introduced OSA commands in version 2.1 of HyperTalk. These were limited to making HyperCard a client application capable of sending any of the "Required" events to a server application. Much more exciting was the release of AppleScript in 1993. This was a completely new scripting language, based loosely on HyperTalk, but supplied with it's own "Script Editor" application. The Script Editor allows users to not only write their own scripts, but also provides a macro record facility (similar to many spreadsheets) so scripts can be created implicitly. The Script Editor also has a "Dictionary" function, which lists the events a selected application will respond to.

It should be noted that since version 2.2, HyperCard can make full use of scripting, by allowing stacks to contain both HyperTalk and AppleScript commands.

High Level Communication in Windows Using DDE

DDE was designed as a way of passing data between Windows applications. Rather than being restricted to static data in a file, applications supporting DDE allow their users access to data, calculated and supplied dynamically by another application. Suggested uses for DDE include:

linking applications to real time data such as the stock exchange or, creating compound documents, where a word processor chart is connected to spreadsheet data.

Within the context of the chart example, users would first state the name of the spreadsheet application to provided the data. The word processor would then query all currently executing, DDE compliant, applications until the required spreadsheet responds. The two applications establish a communication channel, allowing the user to include the foreign application's name, when specifying a path to the chart's data source.

To fully support the dynamic exchange of data, the DDE protocol includes commands for file and data manipulation. A client DDE application can request that its server close, open and save data files, or perform calculations on a given piece of data. It is this aspect of DDE which can be exploited to provide integration of Windows applications. Unfortunately, Windows has no equivalent to AppleScript, prohibiting novice programmers access to this group of application facilities. To make full use of DDE, the programmer must learn a general purpose programming language, such as Visual Basic or C. Obviously this makes DDE of little use to anyone except experienced programmers.

Case Study

The software described in the case study has been implemented on both Macintosh and Windows platforms. Two of the goals for this implementation were that: (1) The software should perform identically on both platforms; (2) The integration of packages should be seamless, requiring no extra input from the user. In aiming to satisfy these goals, the implementation effort identified many types of problem involved with the high level integration of software packages.

Situation

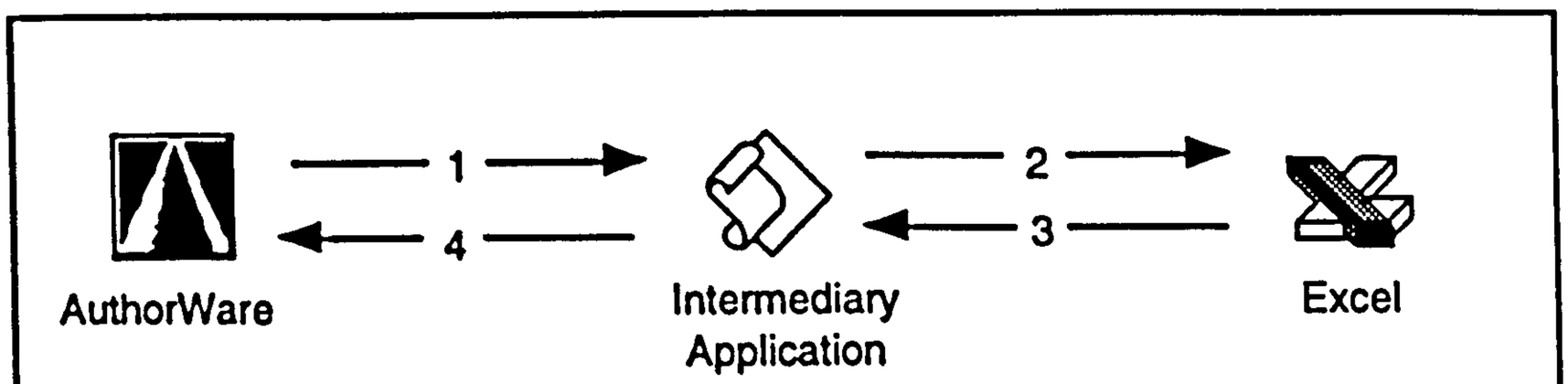
The Mathematics Department at Stirling was involved in the United Kingdom Mathematics Courseware Consortium (UKMCC) project, producing CAL software for university mathematics degree tuition. The design specification for this software requires simultaneous use of both Microsoft Excel and AuthorWare. AuthorWare is launched first and is used to give the student information on a topic of interest. Once the required information has been imparted, the AuthorWare window shrinks to fit the leftmost half of the screen and presents the user with an Excel spreadsheet in the rightmost half of the screen. The spreadsheet contains data which the student then manipulates, using instructions provided in the AuthorWare window. After a few similar sessions, the student is finally presented with a test. In this case, the AuthorWare window contains a collection of questions about properties of a data set, which has just appeared in yet another new Excel spreadsheet. The student must then use the skills they have gained in previous sessions to answer the questions displayed by AuthorWare ; their score being saved to disk once the test is complete.

Problem 1 - Opening and Closing Excel Documents

The first limitation to be overcome was AuthorWare's inability to communicate with other applications. The only command available is *JumpOutReturn*, which opens an application with a specified file, returning to AuthorWare once the application finishes execution. Initial experiments with this command revealed that it could be used to open Excel spreadsheets, provided Excel was not already executing. If Excel was running, however, the request to open a specified spreadsheet file was simply ignored. Clearly, an alternative solution had to be found.

Macintosh Solution

Although the current version of Macintosh Excel provides excellent support for Apple Events, AuthorWare on the Macintosh can only respond to the required suite. It then becomes impossible to write some higher level Apple Script program which controls the interactions of both packages. The further consequence is that AuthorWare cannot directly send an "open document" or "close document" event to Excel. It is possible, however, to create an intermediary application with the Script Editor which will cause Excel to open or close the desired spreadsheet. This application can then be called using JumpOutReturn, with the added benefit that Excel remains active whilst control is automatically passed back to AuthorWare (the intermediary application terminates after passing the event to Excel). See **Figure 1:** Interaction between Excel and AuthorWare.



Windows Solution

Although AuthorWare was not designed to use DDE, this facility is provided in the form of an external code module (DLL) which adds DDE commands to the AuthorWare command list. There is no documentation supplied for these commands, leaving the programmer to derive command syntax and functionality by looking through the one example program and scanning the code files for the DLL. This does not make for confident use of the DDE facilities, and gives the programmer little idea of the range of DDE commands to which AuthorWare can respond. Fortunately, Excel's level of support for DDE is mirrored by its level of support for Apple Script on the Macintosh. Using the code in the DDE example as a template, it was possible to write AuthorWare code which sent the open and close document events to Excel.

Problem 2 - Support for Student Test

In providing software support for the student test, there are actually two problems to be overcome. The first is asking Excel to remotely execute a macro which calculates the answer to the test whenever the test spreadsheet is opened. Fortunately Excel allows spreadsheets to specify macros to run when the spreadsheet is opened. This effectively absolves AuthorWare of sending any event other than a simple "Open Document."

The second problem lies in passing the test answers from Excel to AuthorWare. Although DDE could be used to pass the results into AuthorWare for Windows, the Macintosh version is not scriptable and, therefore, cannot receive data dynamically. The only workable solution remaining is for Excel to write the test answers into a data file and have AuthorWare retrieve them when necessary. In

an effort to keep the software similar across different platforms, the temporary data file solution was adopted for both the Windows and Macintosh environments.

Problems Unique to Windows

Both AuthorWare and Excel use the Windows Multiple Document Interface (MDI). This means that each application provides its own desktop, showing only icons and windows relevant to that application. Whilst this works well when the user only needs to see one application at a time, it is awkward to display both AuthorWare and Excel simultaneously in separate halves of the screen. Normally, the user would resize the backmost, MDI, window of the application to occupy the required area on the screen. Obviously, the desire to remove any extra burden from the user would preclude this as a viable solution. To resize the application's MDI window automatically may be possible through DDE, but the documentation for each application's DDE facilities was less than comprehensive. Currently, the only workable solution available is to directly alter the application's ".INI" files. This is far from satisfactory.

General Comments.

Apart from the common limitations of these systems (such as applications incapable of supporting the communication protocols), they each have their own unique shortcomings. DDE was intended to be embedded within an application, alterable only by those with access to the source code. Whilst this prohibits unwanted tampering, it also makes it hard for non-programmers to alter the application linking. If Excel was not available and Quatro Pro was used instead, as the case was in one of our laboratories, then the code would need to be altered by someone who knows both the AuthorWare code structure and the DDE commands of Quatro Pro. Apple overcomes these problems by imposing a common protocol standard, and making scripts editable by anyone in possession of a script editor. Having scripts external to applications, however, leaves open the possibility of accidental deletion or forgetting to copy all necessary scripts. This also makes AppleScript unsuitable for applications where security and data integrity is important.

On the whole, this type of programming is ill defined. If the applications to be used actually support the scripting system, then, more often than not, the level of this support is not described within the application's documentation. In creating educational software, however, it can prove to be an invaluable tool in overcoming limitations and communication problems with high level software and fourth generation languages.

Conclusions

To make full use of components such as XCMD's and DLL's, it is necessary to have some knowledge of a programming language, normally C, which can be used to create new components. Code written in this form can then be easily transferred and re-used in a wide variety of software, making better use of development resources.

If combining resources from generic applications, then it is first necessary to learn the communication protocol of the applications involved. To be successful, the host machine must have the hardware resources necessary to multi-task several large software packages. At present, communication protocols are poorly supported and it may be that the desired facility may not be supplied by the server application. It is hoped that the continued development of systems such as OpenDoc will produce small applications which lend themselves to being combined into a single piece of software.

References

[O'Connor 1996] J. O'Connor (1996). http://www.groups.dcs.stand.ac.uk/~history/Mathematical_MacTutor.html

[Macromedia 1997] http://www.macromedia.com/software/authorware/features.html

[Apple 1996] http://product.info.apple.com/productinfo/datasheets/as/Hypercard2.3.html

[Apple 1993] Apple Computers (1993). HyperCard Script Language Guide - Appendix A.

[Feldman 1993] Fieldman, J. (1993). Using Visual Basic 3. Que Corporation.

[Udell 1994] Udell, J. (1994). Component Ware. Byte, 19 (5), page 46

[Udell 1993] Udell, J. (1993). Fighting Fatware. Byte. 18 (5), page 98

[Apple 1997] http://www.opendoc.apple.com/

[Sun 1997] http://java.sun.com/

[IBM 1997] http://www.software.ibm.com/clubopendoc/odvsole.html

Flattering to success

♦ *A Rolls-Royce of a book is* The Moths and Butterflies of Great Britain and Ireland, Volume 3, Yponomeutidae to Elachistidae. *With this multi-authored volume, the monographic treatment of the Lepidoptera enters the home straight towards completion. Beautifully produced, carefully edited, well illustrated and comprehensive in coverage, these are volumes aimed to satisfy the dedicated enthusiast. The 452 pages of volume 3 cover 242 of the smaller caterpillars, with 226 colour pictures of adult moths, illustrations of diagnostic genital features for all species, and 120 or so illustrations of larval cases. Life-history information for each species is extensive, with many details not previously recorded in print, and there are up-to-date and detailed British distribution maps for each species. Published by Harley, edited by Maitland Emmett, £37·50, ISBN 0946589569.*

Get A-life

◆ *From cellular automata to the role of non-hierarchical non-deterministic systems in education,* Artificial Life *surveys a field about to differentiate into non-communicating specialisms. Catch it now, while you can find the philosophy of artificial life and the life of artificial philosophers in one volume. Originally reviewed in hardback by Roger Lewin on 16 September, 1995. Published by Bradford/MIT Press, edited by Christopher Langton, $22·50, ISBN 0262621126.*

Knowledge of bodies

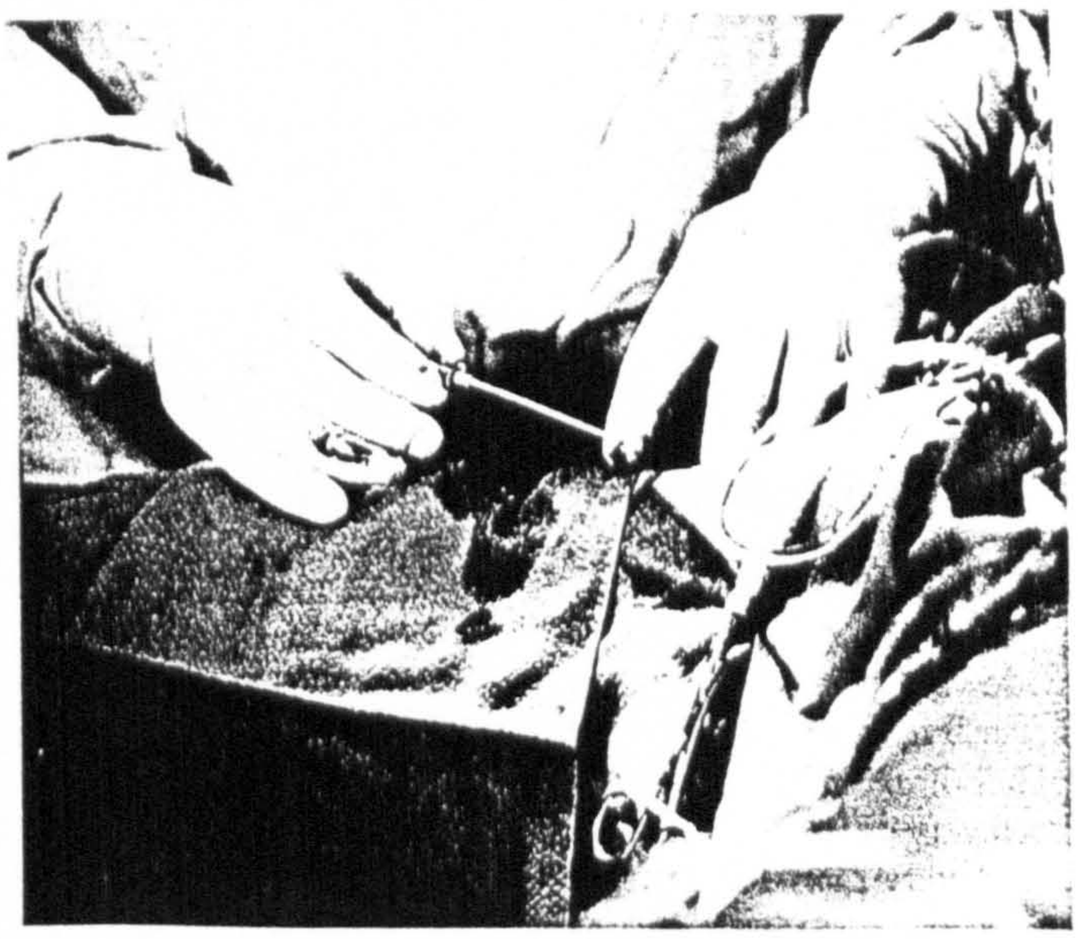
The Wisdom of the Body by Sherwin Nuland, Chatto & Windus, £16·99, ISBN 070116672X

IF YOU have ever wondered why surgical operations are carried out in an "operating theatre" rather than, say, an "operating room", *The Wisdom of the Body* by Sherwin Nuland has some answers. This is the surgeon as theatrical superstar, the surgeon as hero, the surgeon snatching patients from the jaws of death with deft swipes of his scalpel. In places it resembles a string of episodes of *ER*, retold by a rather grand and incorrigible old thespian, who intersperses his wordy narrative with plenty of philosophical asides, bons mots and literary quotations.

Aristotle, Freud, Shakespeare, Rabelais, Nietzsche, Emerson, Auden, Leibnitz and Pope are all invited to put in their two-pennyworth. Elsewhere, the book includes solid chunks of medicine or biology, explained with clarity and sewn as neatly as a good suture into the surgical sagas.

The book covers a huge territory, from the details of DNA replication and the discovery of neurotransmitters to the cause of sickle-cell anaemia and the cooking methods used for traditional pig feasts in the New Guinea highlands. It deals, in passing, with Shelley's *A Defence of Poetry*, the ritual preparation of kosher meat, evolutionary speculation about the orgasmic experiences of Neanderthal females, the history of vitalism and the author's feelings on the birth of his first child. Perhaps inevitably, given this vast terrain, the literary style veers from cool scientific descriptions to grandiose descriptive passages that seem to belong to another century, and then to casual, streetwise, occasionally even slangy language.

The philosophical viewpoint changes just as unpredictably. In places Nuland expresses a reductionist view of the most extreme sort, such as "It had long been known that small differences in proteins are the reasons for the differences between various species." Elsewhere, he swings into an equally extreme kind of biological mysticism. There is much about Man (sic) being only a little lower than the angels and "fearfully and wonderfully mad". For all its faults this is, however, a wonderful book. The surgical dramas will keep you turning the pages compulsively. □



M. Septimus/Photonica

Linda Gamlin is a science editor

Bill's back room

FANCY an Internet or intranet server for your PC or Mac network? You might like to try BackOffice, an attempt by Microsoft to win a share of the server software market. Built on the increasingly popular NT 4·0 Server, BackOffice also includes database (SQL), e-mail/groupware, Internet, SNA and proxy servers along with Microsoft's own remote server management software.

The strength of this package is the relatively seamless way in which it works. The Internet Information Server (IIS) supports HTTP, Gopher and FTP protocols, all using the security model of the underlying operating system. Also included in the package is Microsoft's FrontPage HTML editor.

The package is expensive at £768 (educational) and £1549 (commercial), but it is intended to solve serious networking problems. Given that IIS is bundled free with NT 4·0 Server and Workstation, BackOffice may be overkill for most Net publishing tasks. If, however, you need to install a network server for your company and you have the money and 564 MB of disk space to spare, it offers an attractive solution.

Documentation for NetOffice is provided on CD-ROM with a few slim paper manuals to guide installation. *The BackOffice Survival Kit* from SAMS publishing (£159·95, ISBN 067230953X) includes four thick volumes which cover: SQL Server; Exchange Server, NT Server and a guide to BackOffice administration.

Bundled with these books come five CD-ROMs containing utilities and demonstrations of third-party solutions and enhancements of BackOffice (not to mention the text of two further books on Web site construction and IIS). □

Gary Marsden researches computers at the University of Middlesex

Bibliography

[ABC+84] M.P. Atkinson, P. Bailey, W.P. Cockshott, K.J. Chisholm, and R. Morrison. Progress with Persistent Programming. Universities of Edinburgh and St. Andrews, 1984.

[Aim97] Aimtech. Icon Author Product Information, 1997. http://www.aimtech.com/iconauthor/.

[All88] L. Allison. Some Applications of Continuations. *The Computer Journal*, 31:9–11, 1988.

[All97] Allegant. Supercard Product Information, 1997. http://www.allegant.com/.

[AS89] M.D. Apperley and R. Spence. Lean Cuisine: a low fat notation for menus. *Interacting with Computers*, 1(1), 1989.

[AS95] P. Alley and C. Strange. *ResEdit Complete.* Addison-Wesley, 1995.

[Asa87] P. Asante. Editing Graphical Objects Using Procedural Representations. Technical report, Computer Systems Laboratory, Stanford University, 1987.

[Bac90] J. Backus. Can programming be liberated from the Von-Neuman style? In *ACM Turing Award Lectures.* Addison-Wesley, 1990.

[Bar86] N. Baron. *Computer Languages.* Pelican, 1986.

[Bea93] P. Borenstein et al. *Think Class Library Guide - Version 6.* Cambridge Press, 1993.

[Bla97] H. Blanken. Requirements of a multimedia database. In *Multimedia Databases in Perspective.* Springer, 1997.

[Bor81] A. Borning. The Programming Language Aspects of ThingLab. *ACM Transactions on Programming Languages and Systems*, 3:353–387, 1981.

[Bro82] F. P. Brooks. *The mythical man-month.* Addison-Wesley, 1982.

[Bux83] W. Buxton. Lexical and Pragmatic Considerations of Input Structures. *Computer Graphics*, 1:31–37, 1983.

[BV94] F. Bodart and J Vanderdonckt. On the Problem of Selecting Interaction Objects. In *Proceedings HCI'94*, pages 162–178, 1994.

[CH86] B. Cox and B. Hunt. Objects, Icons and Software IC's. *BYTE*, pages 161–176, August 1986.

[CH93] M. Carlsson and T. Hallgren. FUDGETS: A Graphical User Interface in a Lazy Functional Language. In *ACM - Conference of Functional Programming*, pages 321–330, 1993.

[Coc90] G. Cockton. Lean Cusine: no sauces, no courses! *Interaction with Computers*, 2:205–226, 1990.

[Com92] Apple Computer. *Inside Macintosh: Macintosh Toolbook Essentials*, 1992.

[Com93] Apple Computer. Electronic Guide to Macintosh Human Interface Design, 1993. Addison-Wesley CD-ROM.

[Com95] Apple Computer. HyperCard 2.3 Product Information, 1995. http://product.info.apple.com/.

[Com97] Apple Computer. Apple Newton, 1997. http://www.newton.apple.com/.

[Con95] DSDM Consortium. *Dynamic Systems Development Method.* Tesseract, 1995.

[cor95] NS corporation. *NS Basic Handbook*, 1995.

[Cor96] SoftQuad Corp. *HoTMetaL User Manual.* SoftQuad, 1996.

[Cor97a] Borland Corp. C++, 1997. http://www.borland.com/borlandcpp/.

[Cor97b] Borland Corp. Delphi, 1997. http://www.borland.com/delphi/.

[CT92] G. Coulouris and H. Thimbleby. *HyperProgramming.* Addison-Wesley, 1992.

[Cur81] B. Curtis. A review of human factors research on programming languages and specifications. *Communications of the ACM*, 1:212–217, 1981.

[Cyp91] A. Cypher. EAGER: Programming Repetitive Tasks by Example. In S. Robertson, G. Olson, and J. Olson, editors, *CHI 91*, pages 33–39. Addison-Wesley, 1991.

[dB84] B. du Boulay. Fatal error in pass zero: how not to confuse novices. *Behaviour and Information Technology*, 3:109–118, 1984.

[dB93] E. den Brok. Create Your Own World and See That it is Good. Technical report, University of Nijmegen, June 1993.

[dBFM92] D. de Baar, J. Foley, and K. Mullet. Coupling Appliction Design and User Interface Design. In *Proceedings CHI'92*, 1992.

[dBOM81] B. du Boulay, T. O'Shea, and J. Monk. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14:237–249, 1981.

[Der92] M. Dertouzos. *The User Interface is The Language*, chapter 2, pages 21–30. Addison-Wesley, first edition, 1992.

[DHF96] A. Dearle, D. Hulse, and A. Farkas. Operating Support for Java. In *Proceedings of First International Workshop on Persistence and Java*, 1996.

[Dij68] E. Dijkstra. GOTO Statement Considered Harmful. *Communications of the ACM*, March, 1968.

[Dij75] E. Dijkstra. How do we tell truths that might hurt? In *Selected writings on computing*. Springer-Verlag, 1975.

[dRCS84] J. des Rivieres and J. Cantwell Smith. The Implementation of Procedurally Reflective Languages. *Communications of the ACM*, 8:331–347, 1984.

[DT80] L. P. Deutsch and E. A. Taft. Requirements for an Experimental Programming Environment. Technical report, Xerox PARC, 1980.

[FBB92] B. Freeman-Benson and A. Borning. Constraint Imperative Programming Languages for Building Interactive Systems. In *Languages for Developing User Interfaces*. Jones and Bartlett, 1992.

[FJ89] B. Foote and R. Johnson. Reflective Facilities in Smalltalk-80. In *OOPSLA'89 Proceedings*, 1989.

[FKKM89] J. Foley, Won Chul Kim, S. Kovacevic, and K. Murray. Defining Interfaces at a High Level of Abstraction. *IEEE Software*, 6(1):25 – 32, January 1989.

[Fla96] D. Flanagan. *JavaScript: The Definitive Guide*. O'Riley, 1996.

[FLGD87] G. Furnas, T. Landauer, L. Gomez, and T. Dumais. The Vocabulary Problem in Human-System Communication. *Communications of the ACM*, 30:964–971, 1987.

[Fol86] J. Foley. A User Interface Designer's Aide. Technical report, George Washington University, 1986.

[GA84] L. Gilman and R. Allen. *APL: An Interactive Approach.* New York: John Wiley, 1984.

[Gav91] W. Gaver. Technology Affordances. In *CHI 91*, pages 79–84. Addison-Wesley, 1991.

[GF92] D. Gieskens and J. Foley. Controlling User Interface Objects Through Pre- and Post-conditions. In *CHI'92 Proceedings*, 1992.

[Gil86] D.J. Gilmore. Structural Visibility and Program Comprehension. In *People and Computers: Designing for Usability*, 1986.

[Goo93] D. Goodman. *Mastering Applescript.* Addison-Wesley, 1993.

[Gre77] T.R.G. Green. Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology*, 50:93–109, 1977.

[Gre80] T.R.G. Green. IFs and THENs: Is Nesting Just for the Birds? *Software — Practice and Experience*, 10:373–381, 1980.

[Gre85] M. Green. Design Notations and User Interface Management Systems. In *User Interface Management Systems.* Springer-Verlag, 1985.

[Gre90a] T.R.G. Green. The Nature of Programming. In J.M. Hoc, T.R.G. Green, R. Samurcay, and D.J. Gilmore, editors, *Psychology of Programming*, pages 21–44. Academic Press, 1990.

[Gre90b] T.R.G. Green. *Programming Languages as Information Structures*, pages 117–137. Academic Press, 1990.

[Gro81] Xerox Learning Research Group. The Smalltalk-80 System. *BYTE*, pages 36–48, August 1981.

[Gro97a] Berkley Sather Group. Sather 1.1, 1997. http://www.icsi.berkley.edu/sather.

[Gro97b] Napier Research Group. Napier-88, 1997. http://www-ide.dcs.st-andrews.ac.uk/info/Napier88.html.

[GS95] M. Goodland and C. Slater. *SSADM Version 4 — A Practical Approach.* McGraw-Hill, 1995.

[Hab73] A.N. Habermann. Critical comments on the programming language Pascal. *Acta Informatica*, 3:47–57, 1973.

[Hal97] T. Halfhill. Goodbye GUI, Hello NUI. *Byte*, 22:60–72, 1997.

[Hay84] P.J. Hayes. Executable interface definitions using form-based interface abstractions. Technical report, Carnegie-Mellon University, 1984.

[HC88] R.C. Holt and J.R. Cordy. The Turing Programming Language. *Communications of the ACM*, 31:1410–1423, 1988.

[Hen86] P. Henderson. Functional Programming, Formal Specification and Rapid Prototyping. *IEEE Transactions of Software Engineering*, 12:241–249, 1986.

[Hil92] D. Hils. Visual Languages and Computing Survey: Data Flow Visual Programming Languages. *Journal of Visual Languages and Computing*, 3:69–101, 1992.

[HJ94] P. Hudak and M. Jones. Haskell vs. Ada vs. C++ vs. Awk vs. ... Technical report, Yale University, Department of Computer Science, 1994.

[HMSS88] R. Hill, J. Miller, A. Schulert, and D. Shewmake. UIMS's: Threat or Menace? In J. Rosenberg, editor, *CHI 88*, pages 197–200. Addison-Wesley, 1988.

[Hoc83] J.M. Hoc. Analysis of beginner's problem solving strategies. In *The Psychology of Computer Use*. London: Academic Press, 1983.

[Hor88] B. Horn. An Introduction to Object Oriented Programming, Inheritance and Method Combination. Technical report, Carnegie Mellon University, 1988.

[HS91] H. Hartson and E. Smith. Rapid prototyping in human-computer interface development. *Interacting with Computers*, 3:51–91, 1991.

[HS95] S. Houde and R. Sellman. In Search of Design Principles for Programming Environments. In *CHI'95 Proceedings*, 1995.

[IBM97] IBM. Holt Group - Home Page, 1997. http://www.software.ibm.com/clubopendoc/.

[Inn97] Neural Innovation. Proforma Neural Modelling, 1997. http://www.neural.co.uk/.

[Int97] Digital Technologies International. Facespan - Home Page, 1997. http://www.facespan.com/.

[JM] M. Jones and G. Marsden. Overcoming the limitations of the small screen. IEE colloquium presentation - Savoy Place, May 1997.

[JNZM92] J. Johnson, B. Nardi, C. Zarmer, and J.R. Miller. ACE: A new approach to building interactive graphical applications. Technical report, Hewlett Packard, 1992.

[Joh93] J. Johnson. ACE: Building Interactive Graphical Applications. *Communications of the A.C.M.*, 36(4):41–54, 1993.

[KA86] C. Kessler and J. Anderson. Learning Flow of Control. *Human-Computer Interaction*, 2:135–166, 1986.

[Kay82] A. Kay. New Directions for Novice Programming in the 1980's. *Programming Technology*, 2:210–247, 1982.

[Kur81] T. Kurtz. BASIC. In *History of Programming Languages.* New York: Academic Press, 1981.

[Lel88] W. Leler. *Constraint Programming Languages: their specification and generation.* Addison-Wesley, 1988.

[Lev84] S. Levey. *Hackers.* Penguin, 1984.

[LH87] J. Layman and W. Hall. Logo: A Cause for Concern. Technical report, Dept. of Computer Science, Southampton University, May 1987.

[Lin95] D. Linthicum. The End of Programming. *Byte*, 8, 1995.

[LK90] D. Lau-Kee. *Visual and By-Example Interactive Systems for Non-Programmers.* PhD thesis, Dept. of Computer Science, University of York, January 1990.

[LWSS80] H. Ledgard, J. Whiteside, A. Singer, and W. Seymour. The Natural Language of Interactive Systems. *Communications of the ACM*, 23:556–563, 1980.

[Mac91] W. Mackay. Triggers and Barriers to Customizing Software. In Scott Robertson, Gary Olson, and Judith Olson, editors, *CHI 91*, pages 153–190. Addison-Wesley, 1991.

[Mac97a] Macromedia. Macromedia AuthorWare, 1997. http://www.macromedia.com/software/authorware/.

[Mac97b] Macromedia. Macromedia Director, 1997. http://www.macromedia.com/software/director/.

[Mae87] P. Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA'87 Proceedings*, 1987.

[Man97] B. Manners. Computer Museum, 1997. http://swift.eng.ox.ac.uk/.

[Mar86] J. Martin. *Fourth Generation Languages.* MacMillan Press, N.Y., 1986.

[Mar91] J. Martin. *Rapid Application Development.* MacMillan Press, N.Y., 1991.

[Mar92] G. Marsden. Investigation into the provision of HyperCard interfaces to Mathematica. Master's thesis, University of Stirling, 1992.

[Mar94] G. Marsden. Designing an Interface Programming Language for the End User. In *Adjunct proceedings of HCI 94*, 1994.

[Mar95] G. Marsden. Overcoming Design and Execute Modes in User Interface Design Environments. In *Adjunct proceedings of HCI 95*, 1995.

[MAS91] R. Mulligan, M. Altom, and D. Simkin. User Interface Design in the Trenches: Some Tips on Shooting From the Hip. In S. Robertson, G. Olson, and J. Olson, editors, *CHI 91*, pages 232–236. Addison-Wesley, 1991.

[Mat97] MathWise. MathWise Business Mathematics Training Package, 1997. http://www.bham.ac.uk/mathwise.

[May81] R. Mayer. The psychology of how novices learn computer programming. *Computing Surveys*, 13:121–141, 1981.

[MC97] G. Marsden and K. Chan. Overcoming Resource Limitations. In *Proceedings of Ed-Media 97*, 1997.

[MCH92] B. Myers, David C., and B. Horn. *Report of the End-User Programming Working Group*, chapter 19, page 344. Jones and Bartlett, 1992.

[Mee81] L. Meertens. Issues in the design of a beginner's programming language. *Algorithmic Languages*, 1:167–184, 1981.

[Met97] Metrowerks. Code Warrior programming environment, 1997. http://www.metrowerks.com/.

[Mey89] B. Meyer. You Can Write, But Can You Type? *JOOP*, pages 58–67, March 1989.

[MGD+90] B. Myers, D. Giuse, R. Dannenberg, B. Vander Zaner, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer*, 11:71–85, 1990.

[Mic90] Sun Microsystems. *OpenLook Graphical User Interface Application Style Guidelines.* Addison-Wesley, 1990.

[Mic97a] Microsoft. Active-X Product Information, 1997. http://www.microsoft.com/activex/.

[Mic97b] Microsoft. Visual Basic Product Information, 1997. http://www.microsoft.com/vbasic/.

[Mic97c] Microsoft. Visual Basic Script, 1997. http://www.microsoft.com/vbscript/.

[Mic97d] Microsoft. Visual C++, 1997. http://www.microsoft.com/visualc/.

[Mic97e] Sun Microsystems. The Java language homepage, 1997. http://java.sun.com.

[Mon86] A. Monk. Mode errors: a user-centred analysis and some preventative measures using keying-contingent sound. *International Journal of Man Machine Studies*, 24:313–327, 1986.

[Mor82] J. Morris. Real Programming in Functional Languages. Technical report, XEROX-PARC, 1982.

[Mor93] R. Morrison. Towards Simpler Programming Languages: S-algol. Technical report, Computational Science Department, University of St. Andrews, 1993.

[MR92] B. Myers and M. Rosson. Survey on User Interface Programming. In *CHI'92 Proceedings*, 1992.

[MVZ92] B. Myers and B. Vander Zanden. Environment for Rapidly Creating Interactive Design Tools. *Visual Computer*, 8(2):94–115, 1992.

[MW] A. Monk and P. Walsh. Browsers for literate programs: avoiding the cognitive overheads. Not published.

[Mye86] B. Myers. Visual Programming, Programming by Example and Program Visualisation; A Taxonomy. In *Proceedings of CHI'86*, 1986.

[Mye87] B. Myers. Gaining General Acceptance for UIMSs. *Computer Graphics*, 21:130–134, 1987.

[Mye89] B. Myers. User-Interface Tools: Introducation and Survey. *IEEE Software*, 6(1):15 – 23, January 1989.

[Mye91] B. Myers. State of the art in User Interface Software Tools. Not published, July 1991.

[Mye92a] B. Myers. *Ideas from Garnet for Future User Interface Programming Languages*, chapter 10, pages 147–157. Jones and Bartlett, 1992.

[Mye92b] Brad A. Myers. *Introduction chapter*, chapter 1, page 1. Jones and Bartlett, 1992.

[Mye94] B. Myers. Challenges of HCI Design and Implementation. *Interactions*, 1(1), Jan 1994.

[Nee95] R. Needleman. Most Important Software Products. *Byte*, 9:64–65, 1995.

[Nie] J. Nielsen. What do users really want? DRAFT: Submitted to International Journal of Human Computer Interaction.

[Nie89] J. Nielsen. Prototyping User Interfaces Using an Object-Oriented HyperText Programming System. In *NordDATA89*, pages 1–6, 1989.

[Nie91] J. Nielsen. The learnability of HyperCard as an object-oriented programming systems. *Behaviour and Information Technology*, 10:111–120, 1991.

[Nie93] J. Nielsen. Noncommand User Interfaces. *Communications of the ACM*, 36:83–99, 1993.

[NL95] W. Newman and M. Lamming. *Interactive System Design.* Addison-Wesley, 1995.

[NM90] B. Nardi and J. Miller. The Spreadsheet Interface: A Basis for End User Programming. Technical Report HPL-90-08, Hewlett Packard, Software Technology Laboratory, March 1990.

[NR94] R. Noble and C. Runciman. Functional Languages and Graphical User Interfaces — a review and a case study. Technical report, University of York, February 1994.

[OA90] D. Olsen and K. Allan. Creating Interactive Techniques by Symbolically Solving Geometric Constraints. In *Proceedings if the ACM SIGGRAPH Symposium on User Interface Software and Technology*, number 3, pages 102–107. ACM, 1990.

[OHS89] C. O'Malley, S. Henessey, and F. Spensely. Experiences with HyperCard as a prototyping tool. Technical report, Institute of Educational Technology, 1989.

[Ols86] D. Olsen. MIKE: The Menu Interaction Kontrol Environment. *ACM Transactions on Graphics*, 5:318–344, 1986.

[Ous96] J. Ousterhout. *tcl and the tk toolkit.* Addison-Wesley, 1996.

[Pea88] G. Pearlman. Software Tools for User Interface Development. In *Handbook of Human-Computer Interaction.* North-Holland, 1988.

[Pem87] S. Pemberton. An Alternative Simple Language and Environment for PCs. *IEEE Software*, 1:56–64, 1987.

[Pfa83] G. Pfaff. *User Interface Management Systems.* Springer-Verlag, 1983.

[Pre87] R. Pressman. *Software Engineering — A Practitioner's Approach.* Software Engineering and Technology. McGraw-Hill, 2nd edition, 1987.

[Psi93] Psion. *Psion Series 3a Programming Manual.* Psion PLC, 1993.

[PvE93] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.

[PWW97] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for Multimedia PCs. *Communications of the ACM*, 40:24–38, 1997.

[PYD91] R. Pauch, N. Young, and R. DeLine. *SUIT: The Pascal of User Interface Toolkits.* Addison-Wesley, 1991.

[RA90] M.B. Rosson and S. Alpert. The Cognitive Consequences of Object-Oriented Design. *Human-Computer Interaction*, 5:345–379, 1990.

[Rea89] C. Reade. *Elements of Functional Programming.* Addison-Wesley, 1989.

[Ree94] J. Reekie. Visual Haskell: A first attempt. Technical report, University of Technology, Sydney, 1994.

[RKS+91] L. Rowe, J. Konstan, B. Smith, S. Seitz, and Chung Liu. *The PICASSO Application Framework.* Addison-Wesley, 1991.

[RS85] B. Ratcliff and J.I.A. Siddiqi. An emperical evaluation investigation into problem decomposition strategies used in program design. *Internation Journal of Man Machine Studies*, 22:77–90, 1985.

[RS93] A. Reid and S. Singh. Budgets: Cheap and Cheerful Widget Combinators. In *Draft Report of the 1993 Glasgow Workshop on Functional Programming*, 1993.

[Ryh87] J. Ryhne. Tools and Methodology for User Interface Development. *Computer Graphics*, April:78–87, 1987.

[San96] J. Sanders. Software Publishers Association Fifth Annual Consumer Survey, 1996. http://www.spa.org/research/releases/press1.htm.

[SE83] E. Soloway and K. Ehrlich. Emperical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, 10:595–609, 1983.

[She81] B. Sheil. The Psychological Study of Programming. *Computing Surveys*, 13:101–119, 1981.

[Shn85] B. Shneiderman. Overcoming Limitations Imposed by Current Programming Languages. In *The Role of Languages in Problem Solving.* Elsevier Science Publishers (North-Holland), 1985.

[Shn93] B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, 8, 1993.

[Shu88] Nan C. Shu. *Visual Programming.* Van Nostrand Reinhold, 1988.

[Sin91] S. Singh. *Using XView/X11 from Miranda*, pages 352–363. Springer-Verlag, 1991.

[Sin92] D. Sinclair. *Graphical User Interfaces for Haskell*, pages 252–257. Springer-Verlag, 1992.

[SKN90] G. Singh, Chun Hong Kok, and Tneg Ye Ngan. DRUID: A System for Demonstrational Rapid User Interface Development. In *Proceedings if the ACM SIGGRAPH Symposium on User Interface Software and Technology*, number 3, pages 108–111. ACM, 1990.

[Som96] I. Sommerville. *Software Engineering; Fifth Edition.* Addison-Wesley, 1996.

[SS92] D. Smith and J Susser. *A Component Architecture for Personal Computer Software*, chapter 3, pages 31–56. Addison-Wesley, 1992.

[SSSW85] B. Shneiderman, P. Shafer, R. Simon, and L. Weldon. Display Strategies for Program Browsing. In *IEEE Conference on Software Maintenance*, 1985.

[Ste87] L. Stein. Delegation is Inheritance. In *OOPSLA Proceedings*, 1987.

[Str68] C. Strachey. *Fundamental Concepts in Programming Languages.* North-Holland, 1968.

[SUC92] R. Smith, D. Ungar, and B. Chang. *The Use-Mention Perspective on Programming the Interface*, pages 79–89. Addison-Wesley, 1992.

[Sut63] I. Sutherland. *SketchPad: A Man-Machine Graphical Communication System.* PhD thesis, MIT Computer Science Department, 1963.

[SW79] W. Strunk and E.B. White. *The Elements of Style.* Macmillan, 1979.

[Sze88] P. Szekely. *Separating the user interface from the functionality of application programs.* PhD thesis, Carnegie Mellon, 1988.

[TCJ92] H. Thimbleby, A. Cockburn, and S. Jones. HyperCard: An Object Oriented Disapointment. Not published, February 1992.

[Tec97] Imperial Software Technology. XDesigner User Interface Builder, 1997. http://www.ist.co.uk/xd/index.html.

[Ten77] R.D. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8:97–112, 1977.

[Ten81] R.D. Tennent. *Principles of Programming Languages.* Prentice-Hall, 1981.

[Thi90] H. Thimbleby. *User Interface Design.* Addison-Wesley, 1990.

[Thi94] H. Thimbleby. View binding and user enhanceable systems. *The Visual Computer*, 10:337–349, 1994.

[Too91] R. Took. The Active Medium: A conceptual and practical architecture for direct amnipulation. In *Proceedings of HCI'91, pp 249-264*, 1991.

[Tur81] D.A. Turner. Functional Programming. Technical report, University of Kent, 1981.

[UTMP93] J. Udell, T. Thompson, R. Malloy, and E. Perratore. Fighting Fatware. *Byte*, 4:98–108, 1993.

[VZ92] B. Vander Zanden. An Active-Value Spreadsheet Model for Interactive Languages. In *Languages for Developing User Interfaces*, pages 183–210. Jones and Bartlett, 1992.

[Wad92] P. Wadler. The Essence of Functional Programming. Technical report, Glasgow Department of Computer Science, 1992.

[Way96] P. Wayner. Java Chips. *Byte*, 21:79–88, 1996.

[WC88] L.B. Wilson and R.G. Clark. *Comparative Programming Languages.* Addison-Wesley, 1988.

[Wir87] N. Wirth. *Turing Award Speech*, pages 179–189. ACM, 1987.

[Wol91] S. Wolfram. *Mathematica - A System for Doing Mathematics by Computer.* Addison-Wesley, 2 edition, 1991.

[WR92] C. Waite and J. Rudolf. Completing the Job of Interface Design. *IEEE Software*, pages 11–22, November 1992.

[XVT97] XVT. XVT toolkit and interface builder, 1997. http://www.xvt.com/.

[ZZ92] J. Zwaan and R. Zwart. Graphics for ABC. Technical report, CWI, Amsterdam, 1992.