ISSN 0254–2757

QI QUÆSTIONES INFORMATICÆ

Volume 5 • Number 3 December 1987

An official publication of the Computer Society of South Africa and of the South African Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereeneging van Suid-Afrika en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

QUÆSTIONES INFORMATICÆ

An official publication of the Computer Society of South Africa and of the South African Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Africa en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

Subscriptions

Annual subscription are as follows:

	SA	US	UK
Individuals	R10	$7	£5
Institutions	R15	$14	£10

Computer Society of South Africa
Box 1714 Halfway House

Quæstiones Informaticæ is prepared by the Computer Science Department of the University of the Witwatersrand and printed by Printed Matter, for the Computer Society of South Africa and the South African Institute of Computer Scientists.

A PROGRAM DEVELOPMENT ENVIRONMENT FOR MICROCOMPUTERS

S.P.Byron-Moore
Department of Computing Science
University of Zimbabwe

ABSTRACT

Any distractions in the working environment can affect the quality and quantity of output. For example, noise may disturb a person's thought processes and lead to errors. In addition, the programmer is subject to other, less tangible, distractions arising from the use of the software tools needed for the development process. This paper examines the latter type of disturbance and discusses ways to minimise its detrimental effect on a programmer's concentration.

Keywords Programming environment, microcomputer software.

1 INTRODUCTION

Although the programming environment has a significant effect on the software development process, few programmers give much thought to its design. Many tolerate an unsuitable environment without realising that it may influence the quality of the code they produce. This paper examines the effect of the programming environment and the factors to consider when setting it up. Finally, it looks at an example system to illustrate some of the points that have been discussed.

2 SCOPE OF ARTICLE

2.1 Type of Programmer

We will look at the type of environment required by a serious program developer working on a microcomputer. We define a 'serious' developer to be a person who wishes to produce efficient programs for long term use. This excludes the 'quick and dirty' programmer, who for one reason or another, 'jumps in the deep end' with little thought for program design and is happy with the first version of his program that produces apparently correct answers. It also excludes the student programmer, who may carry out a lot of program development but mainly produces programs that will only be run once or twice.

We could expect the serious program developer to have a reasonable degree of technical knowledge of the computer system that he is using; it is quite probable that program development comprises a substantial part of his job. However, as his main aim is program development, his environment should allow him to achieve this aim with the minimum of distraction.

2.2 Development Environment

Our main interest is in a program development environment. Such an environment should provide ready access to a good selection of flexible, easy to use software tools. Contrast this with an execution environment which should allow for efficient execution of applications programs. As Dolotta points out, 'program development and execution of the resulting programs are two radically different functions'. For this reason, an environment designed for the joint aims of program development and program execution is unlikely to be the most effective one for both purposes.

3 PROGRAMMING ENVIRONMENT

3.1 Effect of the Environment

In recent years, there has been much interest in ergonomics and the computer workplace. It is

questiones informaticæ, 5, 3, pp 13-20, issn 0254-2757

widely recognised that a worker's efficiency and accuracy can be affected by his physical environment [5,6]. For example, a program developer may suffer from physical distractions such as noise, insufficient ventilation or heating, uncomfortable furniture and interruptions by others.

However, there are subtler intellectual distractions which may seriously disturb his thought processes. For many computer systems, a conscious effort is needed to utilise both the operating system and the software tools. Often, a programmer must refer to a manual for clarification of the use of a particular utility. In many instances, he has to enter information that could have been supplied by a program. Short term distractions like these may disturb the programmer's concentration and lead to errors.

Potentially more serious are long term distractions. For example, if the developer needs a programming aid that is not available, he may be forced to write one himself. This may divert him from his principal task long enough to make him forget his overall development plan, thus leading to bad design.

All distractions divert the developer's attention his main objective — that of producing an efficient, easy-to-use program in a reasonable time span. They may cause frustration as they delay the developer and make his task unduly difficult. Certainly, they waste the developer's time and disturb his concentration.

With suitable planning of the programming environment, many intellectual distractions can be avoided or, at least, minimised. For example, the provision of good on-line help files will reduce the need for reference to manuals, careful design of software tools will ensure that user input is never requested needlessly.

Linhart [2] maintains that 'A good programming environment can affect programming ease and code quality more than you might imagine....If a system is comfortable and easy to use, you will use it more frequently and produce better work with it'. This brings us to another point that of motivation. If a developer finds his programming environment not conducive to software development, he may decide, consciously or unconsciously, to avoid contact with this environment whenever possible.

3.2 Software Tools

Clearly, the development environment must be carefully designed and the software tools themselves must be carefully chosen. As with any other implements, it is important that they should be:

- easy to use
- appropriate
- readily available

To ensure ease of use, it is preferable that they are:

- consistent

To provide a flexible development environment which can be tailored to suit a particular user's requirements, it is desirable that:

- a wide range of tools be available
- each tool perform one task only
- there is a method by which the software tools can communicate

3.2.2 Ease of Use

Software utilities must be easy to use if they are to help the developer without distracting him from his main task.

3.2.3 Appropriateness

An inappropriate implement may slow down the completion of a job and may cause the result to be less desirable. Anyone who has tried painting window frames with a large paint brush will verify this!

3.2.4 Availability

Lack of availability can cause a worker to use an inappropriate tool or to avoid doing the job that requires the use of that tool.

3.2.5 Consistency of Tools

If the user-interface of the utilities is consistent, then a new or infrequently used tool will appear familiar to the developer and he should experience little difficulty in using it.

The user of a consistent set of implements is less likely to make errors. Consider a car driver — in principle, he should be able to drive any car. However, if he drives a car with a different layout of controls to the car he is used to, he is likely to make mistakes, for example, turning on the windscreen wipers instead of indicating a right turn.

In a development environment, software tools should have a consistent method of prompting a user for information. For example, either upper or lower case input should be accepted unless there is a specific need for case-dependent entry.

Consistency is desirable in error reporting, with similar programs reporting errors in a similar way. For example, assume that we have an assembler which finds as many errors as possible in one assembly and reports these to the user as a list of messages of the form:

Line xx Error nn

where xx and nn are integers. Our compiler, on the other hand, detects one error and immediately invokes an editor program, with the cursor positioned at the suspected error location. A message is displayed on the editor screen to indicate the nature of the error, for example,

'"do" expected'

These two error reporting methods are radically different. The compiler program has a communication channel with the editor, the assembler does not. The compiler reports one error at a time, the assembler locates as many errors as it can and reports these all together. Even the format of the error message is different; cryptic in the case of the assembler, less so for the compiler. A developer who uses these two programs in rapid succession will mentally have to adapt to their different interfaces. This tiring and distracting adaptation would have been avoided if both programs had had consistent error reporting methods.

3.2.6 Range of Tools

A development environment must provide a wide range of software tools. A developer will then have the flexibility to select utilities which are both suitable for his work and satisfy his personal preferences.

It is essential that an index of these utilities be provided, otherwise the developer may forget (or not realise) what is available and may needlessly search elsewhere for an appropriate tool. This index should be readily accessible and easy to reference. To retain its value, it must be regularly updated.

One programming environment that provides a wide range of software tools is "The Programmer's Workbench UNIX system" [1]. It was designed to offer a uniform set of tools for program development on certain mini-computers. In the microcomputer world, it is fairly easy to collect a range of utilities at a low cost. MS-DOS and CP/M users, for example, can obtain free public domain software from 'PC-Blue' and 'SIG/M' respectively [10]. In recent years, microcomputer software vendors have reduced their prices and aimed for a high volume of sales. The main problem is how to collect a set of software which satisfies the criteria of section 3.2.1. Software from different suppliers will not have a consistent interface. Establishing communications between programs from different sources may be difficult, if not impossible. A developer has the choice of obtaining software tools which satisfy the requirements as far as possible or producing his own software tools.

3.2.7 One-Task, One-Program

Well known proponents of good programming practice agree that one program should

perform one function. Kernigan and Plauger [3], for example, believe that it is better to produce a set of separate tools which work together, rather than to build a complicated program. The writers of the UNIX system decided to 'Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features"' [4].

The main advantages of 'one-task, one-program' are:

- a program which is designed and written for one specific job is more likely to be efficient than a program which has several functions and tries to be 'all things to all men'
- a small single purpose program is less likely to contain errors than a large multi-purpose program
- it is better to write a new program to do a new job rather than adding extra features to an existing program, since this may lead to performance degradation or the introduction of errors.

The 'one program, one job' approach may not be as arduous as it appears at first sight. Assume that we have a program for job A and we want a second program for job B, where job B is related to job A. If the original program is well written, it may be possible to extract portions of code from it which are relevant to a program for job B. This speeds up the process of writing the program for job B.

3.2.8 Communications between Software Tools

To provide maximum flexibility in the development environment, there must be a means to allow a software tool to pass information to another tool. The idea of communications between programs is not new. It is a central design feature of the UNIX operating system which was originally developed in 1969. More recently, in 1983, William Gates, Chairman of Microsoft Corporation, stated that 'Apple and Microsoft are in agreement that the best solution (i.e. method for software development) is to have multiple products (programs) that can easily pass data back and forth' [7].

If inter-program communications are provided there is no need for large unwieldy programs to be written. Instead, a number of small, efficient single-purpose tools can be provided as suggested in section 3.2.7. The developer can use these separately or combine them if required.

Some operating systems, such as UNIX and MS-DOS 3.0 allow a program to communicate with other programs by means of pipes. If the developer does not have an operating system with this facility, it is still possible for communication to take place between programs. One way of doing this is to pass the information via a disc file. This tends to be slow and is only feasible if a hard disc or a disc drive emulator board is used for the transfer. A better method is to pass the information from one program to another via locations in memory. To do this requires a knowledge of the hardware configuration and the operating system, as both these factors may affect the position of memory locations suitable for such a transfer. If the user is allowed to define the memory locations for information transfer and he is supplied with a list of parameters which can be input and output by each program then it is possible to set up communications links between programs.

4 AN EXAMPLE

4.1 Background

In the Department of Computing Science at the University of Zimbabwe, we have put a considerable; amount of effort into producing a development environment which satisfies the above flexibility criteria and seeks to minimise the intellectual distractions to the programmer. Although the current system is still being perfected, we feel that we have made significant advances.

4.2 Implementation of Compiler/Assembler

During the program debugging/testing stage, a considerable amount of time is spent in the edit-compile loop. This is basically non-productive time, which should be minimised to allow the

developer to get on with the more important task of program testing.

Traditionally, the programmer compiles his program and locates one or more syntax errors. Then he invokes an editor program, while remembering the location and type of each error. For each error, he must position the cursor suitably and make the necessary corrections. This method is slow, error prone and full of distractions.

Firstly, the programmer has to remember or write down the position and nature of each error — a task which diverts his attention from the program he is working on. At this stage, transcription errors can easily creep in. Which programmer has not searched, fruitlessly, for a syntax error in the wrong part of his program, because he wrongly copied down or read back the position of an error?

Next, he has to position the cursor at the error location. This may involve typing an editor command such as 'search' or 'move to line X'. Again, this involves the developer in effort which is peripheral to his program development task. Furthermore, a mistake made while issuing such a command will delay the error removal process.

Hopefully, having removed all the errors indicated by the compiler, the developer must now leave the editor and recompile. The compiler might be invoked by typing a command line or by typing a batch file name (if a batch file has been set up to initiate compilation). The command line may be quite complicated especially if command line options have to be included. Whichever the case, an operating system command has to be typed to start compilation. The input of this command can become a monotonous and time consuming task, especially if many recompilations are needed. Certainly, it can distract the user's concentration.

We were initially impressed with Turbo Pascals' method of error removal [8]. In this system, when an error is detected, the editor is automatically invoked and the cursor is positioned at the error location. This technique seemed to offer a vast improvement on the traditional method described above, as it reduced both the amount of user effort and the time lag in locating an error. However, repeated usage of the Turbo system showed it to be rather restrictive so we began looking at similar, but more flexible methods of error removal. The method that we currently prefer is implemented for an assembler-editor system devised by Ridler [9]. In this system, if a syntax error is detected in an assembly language program, the developer is informed and given three choices:

- to abort the assembly at that point
- to continue the assembly and search for the next syntax error (if any)
- to invoke an editor program, with the editor's cursor positioned at the location of the suspected error

This assembler/editor system runs under CP/M 2.2 and the error row and column are passed from the assembler to the editor via unused locations in high memory. This transfer of information would be simpler using an operating system that provides pipes.

The system removes from the user the distraction of noting the position and type of an error. It saves time and effort in locating the error with an editor. It also retains flexibility by allowing the developer to:

- choose whether he wishes to use the assembler-editor link
- choose the editor he wishes to use as part of the assembler- editor system (providing of course that this editor is able to accept a row and column number for initial cursor positioning)
- locate one error per compilation and immediately edit to correct this, or to collect a batch of several errors together before editing
- ignore an error and continue searching for later errors

An alternative method of error removal would be to scan an entire program, locate all syntax errors and then pass details of all the errors to an editor which then positions the cursor at each error in turn. This would avoid the need to recompile after removing one error and may save time, especially when dealing with long programs.

There are problems with this technique; for instance, how should information about multiple errors be passed to an editor? The compiler could store the position and type of each error in a file. Using a suitable editor (for example, an editor with its own macro-language which allows fairly complex manipulations to take place), it would be possible to read the error data from this file, position the cursor suitably, allowing the user to correct the error and repeat this process until the error file is exhausted. Furthermore, the user can print the error file if so desired.

With this method, spurious errors can be a problem. For example, consider a program with

one syntax error which gives rise to a number of spurious error messages. The programmer is led to examine the site of each spurious error in turn. An annoying waste of time!

As we have no practical experience in handling multiple errors by this means, it is impossible to say whether the advantages of such a method outweigh the disadvantages or vice versa.

4.2.1 Choice of Editor

The choice of an editor program depends very much on personal preference. This is definitely a case where "one man's meat is another man's poison"! An editor is the most frequently used tool in a development environment so it is most important that a programmer is able to choose an editor that he likes. For this reason, a compiler with a built-in editor is not desirable. It would be useful if every editor was supplied with a facility to enable the user to specify the initial position of the cursor. It would then be a simple matter to include it in the type of system described above.

4.2.2 Error Removal

The compiler should be tolerant enough to allow a user to ignore an error and continue with compilation. A developer may wish to ignore an error if he realises that its cause lies elsewhere. He may be uncertain of the reason for a particular error and thus may decide to remove all the obvious errors first and later return to the problem error(s). Turbo Pascal, for example, does not allow the user to ignore an error and continue compilation. If a user cannot spot the cause of an error immediately, he may have to leave his microcomputer and consult a manual. This has to be done before he can proceed to remove other errors.

4.2.3 Experience

Experience with our own editor/assembler system has shown it to be a versatile development aid. Ideally, and with consistency in mind, we would like to develop a similar system for each compiler program that we use.

4.2.4 The Operating System

In many instances, the operating system is a major cause of distraction during program development. If, at each stage in the edit-compile-test process, the programmer has to think about and type operating system commands then his attention will be diverted from the program he is developing. One means of avoiding this distraction is to place a shell around the operating system.

4.3.1 A Menu-Driven Shell

In the Department of Computing Science at the University of Zimbabwe, we decided to use a menu-driven interface to our operating system. Our aim was to make it easy for a developer to issue an operating system command. He should be able to give a command with the minimum of keystrokes (few program developers are trained typists!), without having to worry whether he has the format of the command correct, and certainly without reference to a manual. Overall, he should be able to issue an operating system command without undue fuss and distractions.

One of the main arguments against menu driven systems is that they are unduly rigid, allowing the user little flexibility. To avoid this problem we decided that our interface should:

- allow user definition of the menu hierarchy
- allow user definition of the menu contents
- allow user definition of the action of each menu choice
- provide for user-definable default values

With these provisos, the developer can tailor the menu system to his individual requirements. If his requirements change, it is a simple matter to change the menu definitions accordingly.

4.3.2 Example Menu-Driven System

Our menu system comprises a main menu program and a number of optional submenus. For each of these, the menu display and underlying operating system commands may be defined by the developer using a special set-up program. The number of levels and the relationships between menus at various levels is also decided by the user. In addition, there is a special submenu which allows the user to specify various default values for use by the menu system.

The user selects a menu choice by depressing a single key. When he sets up the menu system, he must specify, for each menu choice, the character to be used to select that choice. Normally, we set up menus to use a mnemonic letter as the menu choice selector. For example, 'C' for compile, 'E' for edit and 'X' for execute. These selectors, if sensibly chosen, aid the user's memory. Numerals may be used as the selector characters if desired but this is not recommended, since it involves greater effort to use a menu which does not take advantage of mnemonics.

The special 'defaults' submenu allows a menu user to set a default pathname. If the menu system requires a file name, the user may type this in its full form or, to save effort, he may type a carriage return, in which case the default values are used. This can save a lot of effort during program development when it is common for the user to deal with one file repeatedly. For example, assume that a developer has typed a program into the file 'b:test.pas'. It is likely that he will have to edit 'b:test.pas' to remove errors and then compile the program stored in that file. Under normal circumstances, this process will be repeated several times and each time the user will have to type in the file name. Using the menu system, the programmer should set the default drive = b, default name = test and default extension = pas. Then, when he invokes the editor or compiler and is asked for a filename, he has only to type a carriage return. This is far less taxing for the user than explicitly typing the filename.

The defaults can be used for part of a file name. If a user is prompted for a file name but only enters part of it, then the other parts are filled in by the menu system, using the default values. For example: if the user is asked to provide the name of a file for compilation and he types in 'test', with no extension or drive descriptor, then the default drive value and the default extension value will be used automatically. This is useful in cases where the developer is working with a number of different files, which are all on the same disc and/or have the same extension.

A problem arises if a user requires a command that is not included in a menu. One solution is to provide a menu choice to return to the operating system prompt. Although we usually include such a menu choice for flexibility, it is not the ideal solution as the user is forced to type the entire operating system command explicitly — the situation that we were trying to avoid. Another alternative is to allow the user to define the contents of the menus. This facility is included in our menu system. However, there are occasions when a user will give a particular command repeatedly in one session and then not use it again for a long time. If it is only needed on a temporary basis, it might not be worth including this command in a permanent menu.

Provision for a temporary menu choice is needed. For this reason, we decided to include a 'default command' in our 'defaults' menu. This allows the user to temporarily specify any operating system command as a 'default command' which can then be invoked with a few keypresses. This facility worked well and we were soon asked to extend it to allow a user to define two default commands. Our current system offers this choice. However, it seems that several default commands would not be excessive.

5 CONCLUSION

The efficient production of high quality software depends on the provision of a good programming environment. Although effort is needed to set this up, a serious program developer should find that it is effort well spent, saving him time, unnecessary exertion and many headaches!

REFERENCES

1. T. A. Dolotta, R. C. Haight, J.R. Mashey, [1978], UNIX Time- Sharing System: The Programmer's Workbench, *The Bell System Technical Journal*, **57**, 6, Jul-Aug, Part 2, 2177-2200.
2. J. Linhart, [1983], Managing Software Development with C, *Byte-the Small Systems*

Journal, **8**, 8, Aug,172-182.
3. B. W. Kernigan, P. J.Plauger, [1976], *Software Tools*, Addison- Wesley,
4. M. D. McIlroy, E. N. Pinson, B. A. Tague, [1978], UNIX Time- Sharing System: Foreword, *The Bell System Technical Journal*, **57**, 6, Jul-Aug, Part 2, 1899-1904.
5. E. Maloney, [1981], Video Dismay Terminals, *Microcomputing*, **5**, 7, July.
6. C. Mackay, [1980], *Human factors aspects of visual display unit operation*, Health and Safety Executive, Her Majesty's Stationery Office.
7. W. Gates, [1983], The Future of Software Design, *Byte-the Small Systems Journal*, **8**, 8, Aug, 401-403.
8. Borland International, [1985], Turbo Pascal Version 3.0 Reference Manual.
9. P. F. Ridler Personal conversations.
10. PC-Blue, New York Amateur Computer Club, Box 106, Church Street Station, NY 10008, USA.
11. SIG/M, Box 97, Iselin, NJ 08030, USA.

NOTES FOR CONTRIBUTORS

The purpose of the journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review artilces and exploratory articles of general interest to readers of the journal. The preferred languages of the journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to:

Prof. G. Wiechers
INFOPLAN
Private Bag 3002
Monument Park 0105
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. Manuscripts produced using the Apple Macintosh will be welcomed. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name and affiliation and address. Each paper must be accompanied by an abstract less than 200 words which will be printed at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Tables and figures should not be included in the text, although tables and figures should be referred to in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Figures should also be supplied on separate sheets, and each should be clearly identified on the back in pencil and the authors name and figure number. Original line drawings (not photocopies) should be submitted and should include all the relevant details. Drawings etc., should be submitted and should include all relevant details. Photographs as illustrations should be avoided if possible. If this cannot be avoided, glossy bromide prints are required.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters; between the letter O and zero; between the letter I, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetic order of the author's name, and cited in the text in square brackets. Journal references should be arranged thus:

1. Ashcroft E. and Manna Z., The Translation of 'GOTO' Programs to 'WHILE' programs., *Proceedings of IFIP Congress 71*, North-Holland, Amsterdam, 250-255, 1972.
2. Bohm C. and Jacopini G., Flow Diagrams, Turing Machines and Languages with only Two Formation Rules., *Comm. ACM*, **9**, 366-371, 1966.
3. Ginsburg S., Mathematical Theory of Context-free Languages, McGraw Hill, NewYork, 1966.

Proofs

Proofs will be sent to the author to ensure that the papers have been correctly typeset and *not* for the addition of new material or major amendment to the texts. Excessive alterations may be disallowed. Corrected proofs must be returned to the production manager within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Only orginal papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.