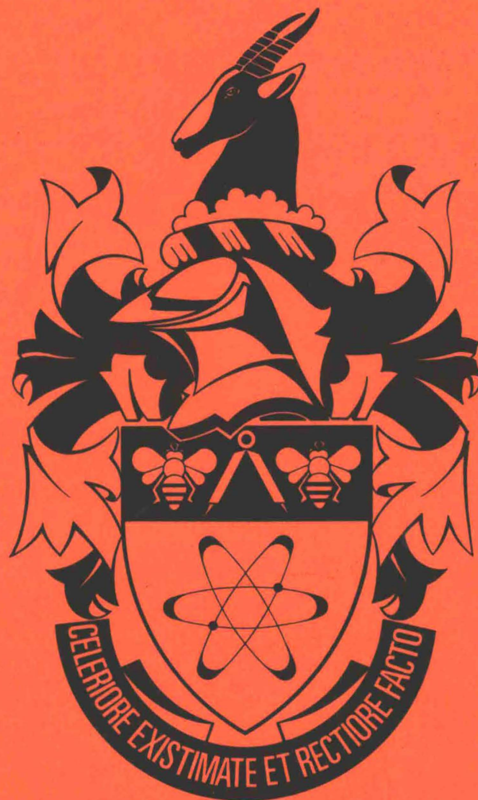


Quaestiones Informaticae

Vol. 2 No. 2 May, 1983

Quaestiones Informaticae

An official publication of the Computer Society of South Africa
'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika

Editor: Prof G. Wiechers
Department of Computer Science and Information Systems
University of South Africa
P.O. Box 392, Pretoria 0001

Editorial Advisory Board

PROFESSOR D. W. BARRON
Department of Mathematics
The University
Southampton S09 5NH
England

PROFESSOR K. GREGGOR
Computer Centre
University of Port Elizabeth
Port Elizabeth 6001
South Africa

PROFESSOR K. MACGREGOR
Department of Computer Science
University of Cape Town
Private Bag
Rondebosch 7700
South Africa

PROFESSOR M. H. WILLIAMS
Department of Computer Science
Herriot-Watt University
Edinburgh
Scotland

MR P. P. ROETS
NRIMS
CSIR
P.O. Box 395
Pretoria 0001
South Africa

PROFESSOR S. H. VON SOLMS
Department of Computer Science
Rand Afrikaans University
Auckland Park
Johannesburg 2001
South Africa

DR H. MESSERSCHMIDT
IBM South Africa
P.O. Box 1419
Johannesburg 2000

MR P. C. PIROW
Graduate School of Business
Administration
University of the Witwatersrand
P.O. Box 31170
Braamfontein 2017
South Africa

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R6	$7	£3,0
Institutions	R12	$14	£6,0

Circulation manager
Mr E Anderssen
Department of Computer Science
Rand Afrikaanse Universiteit
P O Box 524
Johannesburg 2000
Tel.: (011) 726-5000

Quaestiones Informaticae is prepared for publication by Thomson Publications South Africa (Pty) Ltd for the Computer Society of South Africa.

NOTE FROM THE EDITOR

Three points must be made by way of introduction to the second issue of Volume 2 of Quaestiones Informaticae.

Firstly, an apology is in order for the mistake in the date (November 1983 instead of 1982) at the foot of my note introducing the preceding issue. Lacking the services of a professional proof reader, printing errors are bound to show up from time to time, but it is hoped that their number will be kept to a minimum!

Secondly, it is a pleasure to announce that this journal will not only serve to publish papers of a scientific or technical nature on computing matters under the auspices of the Computer Society of South Africa. An agreement has been reached to share the facilities of Quaestiones Informaticae between the CSSA and SAICS, the South African Institute of Computer Scientists. Henceforth this journal will also be used to publish the Transactions of this Institute. This implies certain changes to the cover pages which will be implemented in future issues. I shall continue to serve as editor, but on behalf of SAICS Prof R. J. van den Heever will share some of my duties and act as co-editor.

Finally Mr Edwin Anderssen, of Rand Afrikaanse Universiteit, has agreed to serve as circulation manager for Quaestiones Informaticae. I am grateful indeed that he is willing to serve the journal in this capacity, and look forward to a long period of fruitful cooperation.

G WIECHERS

May, 1983

Detecting Errors in Computer Programs

Bill Hetzel
Blue Cross and Blue Shield of Florida

Peter Calingaert
University of North Carolina at Chapel Hill

Abstract

A controlled experiment was designed and conducted to compare three methods for detecting errors in computer programs: disciplined, structured reading; specification or black-box testing; and a refined form of typical selective testing. Reading was found to be significantly inferior in effectiveness to the other two methods. Specification and selective testing did not differ significantly from each other. On the average, subjects found little more than half the errors present, even on a severity-weighted basis.

Good performance in detecting errors was found to be closely associated with the experimental subjects' computing education, computing background, and self-confidence in performing the experimental tasks. Little association was found with the amount of time spent in detection. The distribution of time to detect the next error was observed to be approximately uniform.

ACM Reviews Categories 4.6; 2.49

Introduction

A major part of the development of any computer program or system is aimed at assuring that the program actually works correctly. The definition of how a program is supposed to work is provided by its *specification*. If the program does not behave as specified, it is said to contain an *error*. Current techniques for detecting errors are far from perfect. A strong incentive is present to develop better methods and improve the effectiveness of our ability to detect errors when they exist. This paper reports on a study [1] and experiment that were conducted to aid in that process.

The basic aim was to learn more about the error-detection activity. Except for some recent work [2] on the classification and tabulation of error data, surprisingly little empirical research has been done in this area. Debugging has been studied by a number of researchers, but the focus has been on the diagnosis and correction of errors once they are found [3,4]. This study was concerned only with the initial detection of errors and not their removal.

It was desired to compare different error-detection techniques, analyze individual differences in using the techniques, develop hypotheses and in general gain as much insight as possible into the error-detection activity. Data were collected experimentally. The basic design of the experiment centered around error-detection sessions in which subjects were given a program and asked to detect the errors in it by using a specified error-detection method. The next section of the paper describes the experiment in detail. A third section presents the results of the experiment and the final section reviews the major conclusions and significance of the study.

EXPERIMENT

Methods

Three methods were selected for comparison — one based on an examination or reading of the program, a second on specification or black-box testing, and a third on a mixture of both reading and testing. The methods were intended to be broadly representative of the spectrum of possible approaches.

In specification testing, the program source code was not available and subjects were given only the specifications. The subjects selected and executed test cases based on the specifications, and then examined the resulting output for discrepancies. This method is sometimes called black-box testing, for the subject has no way to determine the internal construction of the program being tested.

In the mixed method, subjects constructed test cases based upon examination of both specifications and source code. A software tool was provided that indicated execution counts and permitted the subject to ensure systematically that each statement was executed at least once. Every subject but one did in fact execute each statement. Except for this additional criterion, the mixed method is the testing method typically used by most programmers.

The reading method selected was a disciplined and structured desk check. It relied on the fact that the programs were structured to consist of a simple sequence of paragraphs. Each paragraph had the property that flow of control entered at the top and left from the bottom, with the provision that any paragraph could invoke another. The reading procedure consisted of a bottom-up reading and characterization of each paragraph, followed by a top-down reading of the complete program. As each paragraph was read, its external effects were characterized on a special Effects Summary form. Basically, the subject tried to characterize and record the effects of each paragraph as though it were a high-level statement. After all paragraphs were characterized, the program was read top-down with the aid of the Effects Summary form. The resulting program effects were compared with the specifications and any differences recorded.

Programs

The programs selected were actual applications developed at the University of North Carolina (Chapel Hill) Computation Center. One (ANSI) was a program that reads in an arbitrary Fortran program and converts several statements to conform to the ANSI Fortran standard. A second (LIBRARY) was a program to maintain a file of bibliographic references and print a table of contents and a keyword cross-reference index. The third (TEST) was a program to score and evaluate multiple-choice examinations and provide a cumulative examination-grading report.

Each program was written in a typical production-programming environment as a carefully structured PL/I program. The reason for restricting attention to highly structured programs is the expectation that such code is rapidly becoming the norm. The actual errors that were found during the development and production use of the programs were recorded and retained. Each was then reinserted into its program, provided that proper execution of at least one simple test case was possible. The programs were therefore free of syntactic and semantic errors; only logical errors were present.

The descriptions for each program were carefully revised and their clarity was tested in a pilot experiment. The resulting specifications were considerably clearer and more precise than is usually the case. Although all three programs were relatively simple, the modules do represent a spectrum of both applica-

tion and coding complexity. The smallest program (ANSI) contained 75 statements and 5 paragraphs, and the largest (TEST) had 240 statements and 11 paragraphs.

In this manner three program modules were prepared that reflected a production environment and contained naturally occurring errors. Each module executed at least one simple test case correctly and was a realistic approximation to a module that a programmer might have at the start of testing.

Subjects

The aim of this experiment was to find out not so much what programmers actually do, but rather what they *can* do. The attempt was to design an experimental setting to yield results as good as or better than could be expected in actual practice. This meant obtaining subjects as highly qualified and experienced as possible and ensuring that they were strongly motivated. This was achieved by actively recruiting and selecting subjects to participate in the experiment and offering monetary incentive for high performance. Each subject was paid a minimum of $75. An additional payment of as much as $200 was based on relative performance during the experiment. Thirty-nine subjects were selected, most of whom were highly educated and experienced. Six were female and thirty-three were male. Just under half held a master's or Ph.D. degree. Their average work experience in computing was over three years. All had either work experience or academic course experience with PL/I. Their backgrounds are summarized in Table 1.

TABLE 1
Subject Backgrounds

	Minimum	*Mean*	*Maximum*
Age	20	27	38
Grades (A = 3, B = 2, C = 1)	1	2,3	3
Degree (Ph.D. = 4, M.S. = 3, H.S. = 1)	1	2,4	4
Computing Work Experience (months)	6	38	124
Programming Work Experience (months)	3	36	84
PL/I Work Experience (months)	0	18	61
Computer Science Courses	0	8	17
Programming Courses	0	3	9

Administration

The experiment consisted of each subject trying to detect errors in each of the three programs by using a different method for each program. Each subject was randomly assigned to one of three groups A, B, and C. Table 2 shows for each group the correspondence of testing method to program. For example, each subject in group A used the reading method on ANSI, the specification method on LIBRARY and the mixed method on TEST. The order of the three sessions for each subject was also randomized.

TABLE 2
Assignment of experimental conditions

Method		Program	
	ANSI	LIBRARY	TEST
Reading	A	C	B
Specification	B	A	C
Mixed	C	B	A

The basic design of the experiment permitted for the three error-detection methods a comparison that was controlled for differences in subject, differences in program, and differences in order of experimental tasks. The primary performance measure was the percentage of errors found. An alternative measure was a percentage score weighted according to error severity.

All subjects participated in a five-hour instruction session prior to the start of the experiment. The objectives of the experiment and the error-detection methods were explained. The subjects then participated in three error-detection sessions each lasting between three and five hours. Each subject was given a time limit that depended only on the program being verified. The instruction and error-detection sessions were held during the course of one week in a large classroom reserved for the purpose. An experiment staff of four persons coordinated and monitored each session. Generally, two staff members remained in the classroom to supervise and answer questions while the other two submitted and returned test runs. These test runs were prepared on coding sheets given to the experiment staff to be keypunched and run. After execution the output was returned to the subject. Special computer center procedures established for the experiment made it possible to achieve an average turnaround time under fifteen minutes. Each subject was thus provided with excellent response time and freed to concentrate on the error-detection task.

Subjects did not leave the room without being signed out. They were permitted to sign out of the experiment for a break period to avoid losing experiment time waiting for a run to be returned or to assist the staff in keypunching test data. Any time spent on breaks was not included in the subject's time limit, nor counted in his elapsed time. Subjects were instructed to take a break whenever they became fatigued or had to wait for output.

Data

At the start of the experiment, each subject was given a background survey and an attitude survey. The background survey provided data about each subject's education, experience, self-estimates of ability, and other background variables. The attitude survey requested the subject's opinion towards the various methods and the experiment. It was given again after the final error-detection session to permit an analysis of attitude shifts.

The basic data from each session were recorded on error-detection logs. Each subject logged the submission and return of test cases, breaks taken, and suspected errors. At the end of each session, the subject also completed a survey form containing general information about the session. After the experiment, each log was carefully reviewed and encoded. Descriptions of possible errors entered in the logs were matched against the list of actual errors, and the appropriate error number was coded. The coded data were read into a program that reproduced the logs and provided data for further analysis.

RESULTS

Comparison of Methods

The three error-detection methods were compared with respect to the percentage of the total errors detected with each method. The results are summarized in Table 3. Averaged across the three programs, the mixed and specification testing methods detected just slightly fewer than half of the errors present in the programs. For the reading method, only 37 % of the errors were found. Similar relative results are seen for each of the programs individually. The percentages for mixed and specification testing were very close and the percentages for reading were considerably poorer.

TABLE 3
Mean percentage of errors detected by all programmers in each experimental condition

	ANSI	*LIBRARY*	*TEST*	*Mean of 3 Methods*
Reading	48	33	31	37
Specification	55	52	36	48
Mixed	57	48	35	47

An analysis of variance showed the variances accounted for by the programs and by the methods to be highly significant.

The variances due to each group, each replicate, and the different task orders were very small and not significant. The mixed and specification testing methods were not significantly different, but each was very significantly (at the ,001 level) better than reading.

One question was whether these results might be sensitive to the severity of the errors detected. Some errors were very minor and for a few it was even questionable as to whether they were really errors. A number of weighting schemes were used to assign to each error a score based on its severity. The data were then analyzed using the percentage of the total score as criterion. The conclusions were unchanged. The authors could not even think of any plausible weighting scheme that led to a different conclusion. The method differences are present quite uniformly across the different types of errors and are large enough that the choice of weighting scheme has no effect.

The clear conclusion is that the specification and mixed methods are essentially equivalent and that reading is significantly inferior.

Individual Differences

One object of the study was to try to explain the observed individual differences in performance. The ranges of observed differences are shown in Table 4. In general, the best performer was two to three times as capable as the worst in mixed and specification testing, with the spread somewhat greater for reading. To investigate these differences, an analysis was made of the association between each subject's performance and his background, attitudes and approach. Rank order correlation coefficients and χ^2 contingency tables were used as measures of the association between the various variables and a subject's score.

TABLE 4
Extreme percentages of errors detected by all programmers in each experimental condition

		Lowest	*Highest*
	ANSI	20	80
Reading	LIBRARY	7	60
	TEST	0	67
	ANSI	40	67
Specification	LIBRARY	20	73
	TEST	24	72
	ANSI	27	73
Mixed	LIBRARY	27	67
	TEST	24	48

The analyses showed a number of variables to be significantly related to good performance in detecting errors. Regardless of the program being verified or the method used, close association was present between performance and the subject's computing education, computing experience, and self-confidence in performing the experimental tasks.

Little association was found with basic variables such as age, sex, degree level, other self-estimates and attitudes, and the amount of time used by the subject.

One interesting variable that showed moderate association was the number of test cases run by the subject. Particularly in specification testing, there was evidence that some subjects adopted a "try anything" attitude and just created large numbers of test cases in hope that some error might show up. Discounting subjects who submitted over twenty test cases in a single session, the association between good testing performance and the number of test runs was highly significant.

A factor-analysis model of the data was also developed in an effort to explain the underlying relationships in the data. In each of some tens of runs with different variables, about 70 % of the variance in the data was accounted for by variables that could be grouped into four derived factors. The multiple runs tested the model's sensitivity and showed it to be quite stable. The four factors were interpreted as experience, self-esteem, computing education, and attitude toward the experiment. A regression prediction model was also produced. The best-fitting model contained the variables of academic major, education (measured by number of courses taken), self-esteem, attitude, and work experience; it gave an average prediction error of 18 %. The only highly significant variable was academic major. For the group of subjects in the experiment, this variable was closely correlated with computing education and computing experience. Over-all, the regression and factor models support and strengthen the conclusions obtained from looking at the association measures. Subjects with substantial computing education and experience backgrounds who felt confident about their abilities were the ones most likely to do the best jobs of detecting errors.

Other Analyses

A number of other analyses were made in an attempt to develop hypotheses about the error-detection activity and gain additional insight. Two of the more interesting are reported here.

The first was an analysis of the distribution of the time to detect the next error. A program was written to count the detections that occurred in successive time intervals during error detection. Several theoretical distributions were then fit to the counts. The results were surprising. The best fit was simply a uniform distribution, regardless of the method used or the program being verified. Subjects steadily increased their knowledge about the program and tended to try more complex test cases as the session progressed. A plausible explanation for the uniform distribution may be that the increasing knowledge and test case sophistication just about offset the reduced error population. This is a very different situation from the usual program development case. Both Tucker [5] and Schneidewind [6] have reported an exponential increase in error-detection times in that situation.

A second analysis examined the individual errors to see whether different types tended to be found to different extents by the different methods. If at least one-third more of the subjects detected an error with one method than detected it with another, then the error was considered to be found to a significantly different extent by the two methods. Such errors were then categorized in an effort to establish classes of errors that tended to be more difficult to detect by one method than by another. The results confirmed what might be suspected intuitively. Reading did not work well for errors of omission of code statements. Errors involving interrelationships between code segments in different paragraphs were also difficult. Specification testing was more effective for detecting errors that showed up on test cases suggested by the specifications and less effective for hard-to-generate test cass. Mixed testing, which includes some aspects of both reading and specification testing, tended to fall in the middle. Only one error was detected significantly more times as a result of the requirement to execute each of the statements at least once. In general, the path testing requirement seemed to be of very little value.

CONCLUSIONS

How Significant are the Results?

Comparing the three methods gave a very consistent message. Regardless of the performance measure used, specification testing and mixed testing were essentially equal and reading was a poor third. The programs differed significantly, but the relative performance of the methods was the same for each program. In general, this result was unmistakable and convincing.

What about the Generally Poor Performance?

No subject found all the errors in any of the error-detection sessions and, on the average, only about half the errors were detected. Why was this performance so poor? Every effort was made to obtain maximum performance from the subjects. The programs were clearly structured, computer turnaround was excellent, subjects were highly educated and motivated, and they worked without distractions. It is reasonable to conclude that the results are likely to be better than what can be expected in

actual practice and that the detection of errors is a very difficult process. The experiment shows that an intensive period of independent error detection does not provide any assurance of correctness. One might speculate whether financial incentives to program writers and program testers would lead to the generation of fewer errors and the detection of more.

Why wasn't the Mixed Method Better?

The mixed method was designed to have the advantages of both reading and specification testing without the disadvantages of either. That it did not turn out better shows that the time and mental effort required to comprehend the source can be a detriment. Careful concentration on the specifications seems to lead to better test cases and to increase the likelihood of error detection.

How Well can Individual Differences be Explained?

Three groups of variables were found to be significantly associated with error-detection performance — computing education, computing experience, and self-confidence in error detection. Use of test runs was also associated to a point. These associations were found to be present fairly uniformly for all of the programs and methods. The results suggest that many of the skills needed for good error-detection performance can be taught and acquired.

What Else was Learned?

The distribution of the time to find the next error was shown to be approximately uniform. This was attributed to the increasing subject knowledge and submission of more complex test cases offsetting the reduced error population. The experiment also confirmed the logical deductions that reading was not effective for detecting errors of omission and that specification testing was not effective for detecting errors that were hard to generate. Finally, the experiment showed that very little help in detecting errors was provided by the path testing requirements to execute each statement.

Other Contributions

Other benefits of this investigation include the data resource acquired, the experimental methodology, and some suggestions for future research. Worthy of further study are the effect of different numbers of errors in the program being verified, the relationship of program complexity to error detection, and a closer examination of the usefulness of path testing, especially for larger programs. The raw data have been carefully preserved in the first author's dissertation [1].

Acknowledgements

The authors gratefully acknowledge the influence of studies by Gold [7] and Sackman [8] upon the development of the experimental methodology. This work was partially supported by the national Science Foundation under Grant GJ-30410.

References

[1] W.C. Hetzel, *An Experimental Analysis of Program Verification Methods.* Doctoral Dissertation, University of North Carolina at Chapel Hill, Chapel Hill, NC, 1976. (Available from Xerox University Microfilms, 300 North Zeeb Road, Ann Arbor, MI 48106).

[2] B.W. Boehm, R.K. McLean and D.B. Urfrig. Some Experience with Automated Aids to the Design of Large Scale Reliable Software. In: *Proceedings 1975 International Conference on Reliable Software,* IEEE Catalog no. 75CH0940-7CSR, New York, 1975, pp. 105-113.

[3] J. Gould and P. Drongowski. *A Controlled Psychological Study of Program Debugging.* IBM Research Report RC 4083, Yorktown Heights, NY, 1972.

[4] W.C. Hetzel, A Definitional Framework. In: *Program Test Methods,* Englewood Cliffs, NJ, Prentice-Hall, 1973, pp. 7-11

[5] A.E. Tucker, *The Correlation of Computer Programming Quality with Testing Effort.* System Development Corporation Report TM 2219, Santa Monica, CA, Jan. 1965.

[6] N.F. Schneidewind, Analysis of Error Processes in Computer Software. In: *Proceedings 1975 International Conference on Reliable Software,* IEEE Catalog no. 75CH0940-7CSR, New York, 1975, pp. 337-347.

[7] M.M. Gold, *Methodology for Evaluating Time Shared Computer Usage.* Doctoral Dissertation, Alfred P. Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA, 1967.

[8] H. Sackman, *Man-Computer Problem Solving: Experimental Evaluation of Time Sharing and Batch Processing,* Princeton, NJ, Auerbach, 1970.

Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, 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 at:

Department of Computer Science
University of South Africa
P.O. Box 392
Pretoria 0001
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

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

Tables and figures

Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and "open" and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

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 l, the number one and prime; between K and kappa.

References

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

1. ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.
2. BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9**, 366-371.
3. GINSBURG, S. (1966). *Mathematical Theory of context-free Languages,* McGraw Hill, New York.

Proofs and reprints

Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

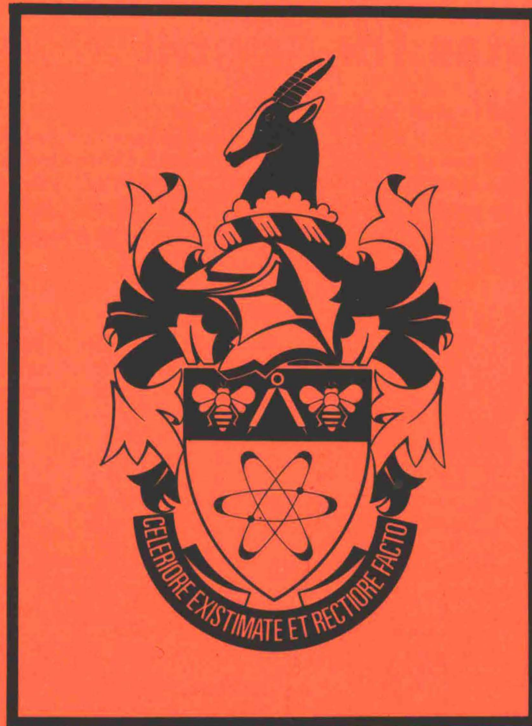
Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

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

Hierdie notas is ook in Afrikaans verkrygbaar.

Quaestiones Informaticae



Contents/Inhoud

Die Operasionele Enkelbedienermodel* 3
J C van Niekerk

Detecting Errors in Computer Programs* 7
Bill Hetzel, Peter Calingaert

Restructuring of the Conceptual Schema to produce DBMS Schemata*..11
S Wulf

Managing and Documenting 10-20 Man Year Projects* 15
P Visser

Data Structure Traces* 19
S R Schach

Case-Grammar Representation of Programming Languages*.......25
Judy Mallino Popelas, Peter Calingaert

Die Definisie en Implementasie van die taal Scrap*...................29
Martha H van Rooyen

*Presented at the second South African Computer Symposium held on 28th and 29th October, 1981.

Printed by Studio Press, Johannesburg