'We are not in a position to measure it.'
(19)

There appear to be three broad reasons for this difficulty:-

i) We have limited appreciation of those properties of software which constitute a quality program. (20)

ii) There are not generally known - or generally accepted - quantitative measures of software quality. (21)

iii) We need to develop firmer foundations for determining the properties and characteristics embodied by software (see Section 7.4.2.4.2 and (22)).

7.4.2.2 Some definitions of Program Quality

Bernstein (23) makes the point that program quality differs significantly from the concept of quality in physical objects. Flaws and errors in programs do not occur spontaneously. They are embedded in the program design, or the code, from its inception. Programs do not malfunction like aging electronic components. This concept may be the underlying reason for the traditional view of trying to define program quality in terms of the number of errors found in the code.

One of those who takes this approach is T.C. Jones. He uses the term 'quality' to mean:-

> '.... the absence of defects that would cause the program to behave unpredictably, or stop successful execution'
>
> (24)

However, there is a body of opinion which suggests that no single measure of program quality (for example 'defects per 1000 lines of source code', Jones (25)) is adequate. A comprehensive list of researchers who advocate a multi-dimensional definition of program quality can be found in McCall et. al. (26). Some opinions will be summarized here.

Knuth (27)

In his Turing Award Lecture of 1974 Knuth discusses the attributes of a 'good' (quality?) program. Initially he follows a traditional line of suggesting that a good program should work correctly to give the desired results and should make efficient use of computer resources. However he adds further dimensions to his definition by suggesting that a good program must not be difficult to change. It now becomes necessary for a good program to be readable and understandable to any person who knows the language in which it is written. Knuth continues that another characteristic of a good program is that it should 'interface gracefully with the User' (particularly when recovering from a human error in input data).

A single measure of program quality will not meet Knuth's definition of a good program.

Hall (28)

In the notes written for the Infotech International lecture on Programmer Productivity, Hall postulates that program quality must be described in terms of the program being developed within time and cost budgets and its efficient execution on the computer without failures. These are somewhat traditional concepts, but he adds further dimensions. He suggests, like Knuth, that a quality program should meet user and design specifications. Hall adds two further points. A quality program should be adequately and effectively documented, and it should be adaptable without rewriting.

In the light of other writers, Hall's definition of program quality may be limited, but his work is part of the trend towards accepting that program quality requires a multi-dimensional definition.

Bernstein (29)

I have already quoted from M. I. Bernstein writing in the Readers Opinion Forum of the April 1979 edition of Datamation. In that article he introduces software quality as a multi-dimensional concept. He makes the point that program quality is more than the number of errors which need repairing, per 1000 lines of source code. What is also necessary is to have an assurance that no action on the User's part will create undesirable results. As can be expected, a quality program should have no defects which will prevent the User getting what is permitted, or will produce erroneous, misleading or unpredictable results.

Perhaps the most significant contribution Bernstein makes in this article is that program quality does not come at just one stage of development (that is, coding or testing), but rather it depends on decisions and choices made throughout the development cycle.

He also claims that program quality is relative to the situation - a 'one-shot' program cannot be judged against the same criteria as a large application system which will run in multiple environments over a 10 year period. (Nor can a batch program be judged against an on-line program which will be executed by hundreds of Users during any one day).

Bernstein suggests that the required quality must be chosen before any program development takes place, and the product must be judged by that choice. Developers can then be evaluated by how well they did in respect of that expectation.

GUIDE Productivity Group (30)

In their document entitled 'Considerations for Productivity Measurement', the GUIDE Productivity Group identify the basic assumptions used in establishing their productivity measurement process. Two of these assumptions are important in the context of defining program quality:-

i) Quality, they claim, is a measure of how good a program is, and therefore must include measurements of the program's

maintainability,
understandability,
completeness,

and the extent to which it conforms to standards.

ii) On the other hand effectiveness is a measure of how appropriate the program is, and therefore is a measure of:-

- how well the program meets the needs of the organization,

- the benefits derived from the use of the program,

- the costs incurred in developing and executing the program.

(30)

Unfortunately, no attempt is made in the document to define the elements of quality in more detail, but again the requirement of a multi-dimensional definition for program quality is apparent.

(It is significant to note that although the GUIDE Productivity Group acknowledge that there will be variances in quality and effectiveness from program to program, they do not try to measure these - in fact they leave this measurement to future efforts).

Gilb (31)

In his Advanced Programming Training Course Gilb quoted freely from the TRW documents where the characteristics of software quality are identified but he did not offer a clear definition of program quality. However, it is possible to deduce some of the elements of what his definition of program quality would contain.

It is possible that the basis for his definition of program quality would be what he calls the 'Program Attribute Specification' Sheet (32). During the course Gilb claimed that unless each relevant attribute was specified for each program to be developed, it is not possible to estimate the cost of developing the program. (Rather overstating the case, he said that without the attribute specification, the program development cost would be between 1 rand and 500 million rand).

Gilb defines 'Program Attributes' as:-

> 'the difference between the various ways of implementing (the minimum requirements of a program)'.
>
> (33)

Unfortunately, what Gilb regards as 'program attributes' must be deduced. In Appendix 'A' of McCall (34) the authors provide a list of software quality factors referenced in the literature, with definitions. Table 7 is an extract of system attributes and definitions quoted or paraphrased from Gilb's writings. Gilb does not appear to make any distinction between system or program attributes, so from this list it becomes apparent that Gilb also subscribes to the view that program quality requires a multi-dimensional definition.

Table 7 A list of system attributes (and their definitions) from Gilb.

Portability - ease of conversion of a system from one environment to another; the relative conversion cost for a given conversion method.

Availability - probability that a system is operating satisfactorily at any point in time, when used under stated conditions.

Reliability - probability that the system will perform satisfactorily for at least a given time interval, when used under stated conditions.

Accuracy - measure of the quality of freedom from error, degree of exactness possessed by an approximation or measurement.

Efficiency - the ratio of useful work performed to the total energy expended.

Maintainability- probability that a failed system will be restored to operable condition within specified time.

Stability - measure of the lack of perceivable change in a system in spite of the occurrence in the environment which would normally be expected to cause change.

Flexibility - the ability of a system to immediately handle different logical situations.

Integrity - probability of system survival when subjected to an attack during a specified time interval.

Generality - degree to which a system is applicable in different environments.

Cost - implementation cost and operational cost.

Time - expected life-span of the system.

Complexity - measure of the degree of decision-making logic within a system.

Precision - the degree to which calculated results reflect theoretical values.

Tolerance - measure of the systems ability to accept different forms of the same information as valid or withstand a degree of variation in input without malfunction or rejection.

Compatability - measure of portability that can be expected of systems when they are moved from one given environment to another.

Repairability - probability that a failed system will be restored to operable condition within a specified active repair time when maintenance is done under specified conditions.

Serviceability - degree of ease or difficulty with which a system can be repaired.

McCall et. al. (35)

One of the objectives of the research undertaken by McCall, Richards and Walters was to determine a set of factors which jointly comprise software quality (36). As one of the steps towards achieving this objective the authors distinguished between software quality factors, and software quality criteria. They defined each of these as:-

> 'Factors - a condition or characteristic which actively contributes to the quality of software.
>
> 'Criteria - attributes of the software, or software production process, by which Factors can be judged and defined.
>
> (37)

After extracting candidate software quality factors from the literature (38), these entries were grouped into a smaller, more concise number of factors which were still regarded as providing a comprehensive definition of software quality.

They found it possible to group these factors into what they called the three 'product activities' of:-

- product operation,
- product revision,
- product transition.

(39)

The factors which they suggest are associated with each of the product activities are as follows:-

i) Product Operation

Correctness	-	Does it do what I want?
Reliability	-	Does it do it accurately all of the time?
Efficiency	-	Will it run on my hardware as well as it can?
Integrity	-	Is it secure?
Usability	-	How convenient do I find it to run?

ii) Product Revision

Maintainability	-	Can I fix it?
Flexibility	-	Can I change it?
Testability	-	Can I ensure it performs as intended?

iii) Product Transition

Portability	-	Will I be able to use it in some other environment?
Reusability	-	Will I be able to reuse some of the software?
Interoperability	-	Will I be able to interface it with other systems?

(Each of these factors is defined in Table 13).

7.4.2.3 The lack of consensus concerning how Program Quality should be measured

The lack of consensus in the industry concerning how program quality should be measured - if it can be measured at all - appears to surpass the disagreement on how programmer productivity should be measured.

The lack of consensus was reflected in the replies received to Questionnaire 1 (see Appendix 'A') which was distributed as part of this research. Answers to the question on whether attempts were being made to measure program quality were received from 42 companies. Half this sample population (21 companies) indicated that they did not try to measure the quality of the programs developed in their environment. The other 21 companies used 17 different combination of methods to assess program quality (see Table 8).

The following observations are made:-

i) Unit Testing (a widely used method of attempting to identify logic errors in programs) can be regarded as a method of measuring program quality. It seems unlikely that a procedure practised as widely as unit testing is excluded from the project development life cycle of the 21 companies who indicated in Questionnaire 1 that they did not measure program quality (although there is a school of thought which suggests that unit testing is too poor a quality assurance method to be worth the investment - see Section 7.4.2.4.1).

Table 8 Methods of Measuring Program Quality used by Installations which answered Questionnaire 1.

Number of program aborts (only)	1
Walk-thru and Inspections (only)	2
Desk-checking by peers (only)	1
Desk-checking by superiors (only)	1
Ease of maintenance (only)	1
Number of program aborts, Walk-thrus, Inspections and Ease of maintenance	1
Number of program aborts, Walk-thrus, Inspections and other	1
Number of program aborts and Desk-checking by superiors	2
Number of program aborts and run speeds	?
Walk-thrus, Inspections and Desk-checking by superiors	1
Walk-thrus, Inspections, Desk-checking by superiors and Ease of maintenance	2
Walk-thrus, Inspections, Desk-checking by peers and other	1
Walk-thrus, Inspections and Ease of maintenance	1
Desk-checking by peers and superiors	1
Desk-checking by superiors and Ease of maintenance	1
Ease of maintenance and Run speeds	1
Other	2

What is more likely is that unit testing is not generally regarded as a quality control procedure, which is surprising. Perhaps it is invalid to regard it as a method of measuring program quality unless the errors identified during unit testing are recorded and analyzed, and the results fed back to the programmers.

ii) It was surprising to find 6 companies (28,6%) of the sample population where the measurement of program quality is attempted, used only one measurement method. Because of the lack of wide-spread success in attempting to control program quality, it would be perhaps more predictable for program implementors to use multiple methods to control the quality of the programs they develop.

iii) Another unexpected result was that 9 (42,9%) of the companies in the sample population claimed they used the Inspection process to control program quality (see Section 7.4.2.4.1). Because the process was not clearly defined in the questionnaire, it is possible that misunderstanding resulted in an inflated figure. It is not common knowledge that this process is widely used.

iv) 'Ease of maintenance' as a measure of program quality was used by 6 (28 6%) of the sample population. The value of this information is severely limited by the lack of a definition of 'maintenance' as used in this context, and what makes it easy.

v) Efficiency seems to be regarded as a facet of program quality by the 2 companies in the sample population who regard 'Run Speeds' as a measure of program quality.

Perhaps part of the lack of consensus reflected in these replies is the result of the companies not having a clear definition of program quality. In Section 7.4.2.2 an attempt was made to identify some definitions of program quality which are found in the literature.

7.4.2.4 Some attempts to measure Program Quality

Attempts to measure program quality appear to fall into two groups:-

- attempts to control quality;
- attempts to specify, measure and control quality in terms of its multi-dimensional definition.

7.4.2.4.1 Attempts to Control Program Quality

Two methods which can be used to control aspects of program quality (but do not provide the mechanism to specify or measure it) are:-

- Unit Testing
- Program Reviews.

A) Unit Testing

This is the most commonly used method to attempt to remove defects from a program. The steps usually carried out at the unit testing phase are:-

- identification of what must be tested;
- design of test cases to include all required tests;
- execution of test runs;
- verification of results obtained.

It seems axiomatic that this process will verify that programs will perform as expected. Sometimes testing aids, like test-data generators or packages which ensure that each instruction is executed, are used in an attempt to improve the defect-removal process.

Unfortunately the process itself has some defects. For example:-

i) Relevant test data is expensive both to create and maintain. M. F. Tighe (41) claims that unit testing phase can double the cost of a product.

ii) The process is intrinsically limited Essentially unit testing can only confirm that the program will execute as expected under the particular set of conditions of the test. It is invalid to deduce anything about the program's accuracy in any other environment.

iii) A second limiting factor is that, at best, unit testing is no more than a defect removal technique. Because program quality is more than the absence of defects (see Section 7.4.2.2), the value of unit testing as a quality control process is limited.

iv) Unit testing is sometimes not appropriate. It is most beneficial for testing self-contained subsystems, but it is not always applicable, for example, to test the enhancement of a program which runs as part of large real-time system.

Because of the limitations of unit testing Gilb (42) suggested that there are alternatives which are more appropriate. He lists these as:-

- the use of programming teams,
- top-down structured development,
- redundant data structures which enable automatic detection and correction of errors in input data,
- dual-coded programs,
- peer checking of programs,
- development of programs specifically for reliability,
- introduction of a number of errors into a program known to one programmer, and the detection of these errors by another programmer.

Unlike Gilb I do not envisage the unit testing phase of the project life cycle being abandoned. However, it is necessary to be aware of the limitations of this approach in controlling program quality.

B) Program Reviews

In answer to the question:-

Why Do We Have Technical Reviews?

Freedman and Weinberg write:-

'The reason we need technical reviews is that although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else .... a review is a way of using the diversity and power of a group of people to:-

a) point out needed improvements in a product of a single person or team,

b) confirm those parts of a product in which improvement is either not desired, or not needed,

c) achieve technical work of more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more manageable.'

(43)

The more widely used methods of program review are the Inspection Team and the Structured Walk-thru (see (44), (45) and (46)). Although these two techniques appear similar, their objectives are sufficiently diverse to require that they be assessed individually. However, this assessment will not be equitable because the Inspection process will be covered in detail while only a summary will be given of the Structured Walk-thru.

a) Structured Walk-thru

Introduced by IBM as part of the so-called Improved Programming Technologies, it appears that a variety of Structured Walk-thrus are used in application development. Although these reviews vary in their objectives and procedures, the following characteristics are usually found:-

i) The Walk-thru is arranged and scheduled by the programmer whose program is to be reviewed.

ii) The reviewers are members of the same project team as the programmer whose work is to be reviewed.

iii) The programmer selects the list of reviewers (although management may well examine the list to ensure that people with the correct mix of skills are included in the reviews).

The more widely used methods of program review are the Inspection Team and the Structured Walk-thru (see (44), (45) and (46)). Although these two techniques appear similar, their objectives are sufficiently diverse to require that they be assessed individually. However, this assessment will not be equitable because the Inspection process will be covered in detail while only a summary will be given of the Structured Walk-thru.

a) Structured Walk-thru

Introduced by IBM as part of the so-called Improved Programming Technologies, it appears that a variety of Structured Walk-thrus are used in application development. Although these reviews vary in their objectives and procedures, the following characteristics are usually found:-

i) The Walk-thru is arranged and scheduled by the programmer whose program is to be reviewed.

ii) The reviewers are members of the same project team as the programmer whose work is to be reviewed.

iii) The programmer selects the list of reviewers (although management may well examine the list to ensure that people with the correct mix of skills are included in the reviews).

iv) Membership of the review team usually excludes management, but the 4 to 6 people who do participate include:-

- the designer of the system - to ensure compatibility and continuity of design;

- developers of other parts of the system;

- people responsible for documenting and testing the system.

v) The reviews are given the material some days (3 or 4 days) before the Walk-thru. They are expected to prepare for the review, and come to the session with a list of questions on points which they feel require investigation or clarification.

vi) A typical Walk-thru is scheduled to last for a specific time - usually no longer than 2 hours. If the review is not complete at the end of the two hours, another Walk-thru is arranged for the next convenient time.

vii) One person is elected to guide the review team through the program. That person (usually the programmer or the designer) compiles a list of all errors, discrepancies, exposures and inconsistencies uncovered during the Walk-thru.

viii) All issues are resolved after the session. The Walk-thru is for problem detection, not problem correction.

ix) A second review could be arranged if a significantly large list of issues has been identified.

Some sources, for example SIMPACT course on Walk-thru procedures, identify 'formal' and 'informal' Walk-thrus. According to their approach, the procedure described above is an 'informal' Walk-thru, while their 'formal' Walk-thru approaches the Inspection Team Process described in the next section. (See discussion on for key differences between the Walk-thru and the Inspection later in this Section).

Provided program quality is closely related to the absence of errors in the code, this review technique has value in controlling program quality.

However, it has already been established (see Section 7.4.2.2) that program quality is more than the absence of defects in the code. Consequently the use of Structured Walk-thru, with its introspective tendencies, will have value in assisting the Project Team to co-ordinate their activities and interface their programs, but will have limited value in controlling program quality. In fact I have not been able to discover any documented evidence of the contribution Structured Walk-thrus have made to improved program quality.

b) The Inspection Team Process

Details of the Inspection Team Process were first documented by Fagan (46) in 1976. He used the technique to control the quality of software programs.

The objective of the Code Inspection (the particular form of Inspections relevant in the context of this research) is to find errors in program code. It needs to be emphasised that, like a Walk-thru, an Inspection is not:-

- to evaluate what is being inspected,
- to judge the programmers who developed the code,
- to identify alternative algorithms to solve the problem,
- a work-shop to correct any errors identified.

Although this appears to be a straight-forward objective, people appear to have difficulty in agreeing on a definition of an ERROR. The definition used at the Standard Bank, where the Inspection Process was introduced in September 1975, is given in Chapter 3. This multi-dimensional definition of 'error' enables the Inspection Process to be considered helpful in controlling program quality.

There is not complete agreement between Fagan (47) and Gilb (48) on the membership of the Inspection Team. At the Standard Bank we tended to follow Gilb in this respect. We found the following factors important:-

i) The Project Team was represented on the Inspection Team by the person responsible for the technical implementation of the system.

ii) The Maintenance Team was represented on all Inspection Teams to help identify the types of errors which cause the most serious maintenance problems.

iii) We discouraged people at Management level from participating in the Inspections because their presence was found to be inhibiting (people feared the Managers were evaluating those who developed the code, as well as looking for errors in the code).

The timing of the Code Inspection is important because Fagan found (49) that holding the Inspection too late in the program development cycle - that is, after the Unit Test stage - was counter-productive. So, each program is inspected after it has been successfully compiled without syntax errors, but before any approach is made to the computer for unit testing. Once the Inspection Team has agreed that the program is of acceptable quality, the Project Team may continue with the unit testing of that program.

Like the Structured Walk-Thru, the Inspection Process has specific phases, but in the more formal approach to Inspections (see 46), each phase has its own specific objective.

Phase 1 - The Overview:
This phase of the process is conducted by the project designer. The objective is to help the members of the Inspection Team who are not part of the Project Team to gain an overall picture of the system being developed, and appreciate how the program to be inspected interfaces with the rest of the system. It is during this Overview that listings of the program are distributed to each member of the Inspection Team.

Phase 2 - Preparation:
The objective of the second phase of the process is to allow the non-project members of the Inspection Team to familiarize themselves with the program to be inspected (see (50) and (51)). This helps to make the error-seeking activity of the next stage of the Inspection as efficient and effective as possible. If flagrant errors are found during this preparation stage they are noted for discussion during the actual Inspection.

Phase 3 - The Inspection Meeting:
The Inspection is normally held 2 or 3 days after the program listings are made available to the non-project members of the Inspection Team.

The objective of the Inspection Session is to find any errors there may be in the program which is being inspected. To assist in the error-finding process, Fagan recommends a set of standard procedures which include:-

- there should be no interruption during the Inspection session;
- the session should not last longer than about 2 hours (should the work be incomplete at the end of the 2 hours, another session should be convened);
- the Project Designer should guide the Inspection Team through the code, ensuring that every piece of logic is covered and each branch is taken;

- each member of the Inspection Team should be adequately prepared, and participate in tne proceedings (inadequate preparation is valid reason for postponing the Inspection).

(52)

After recording (and possibly classifying) all the errors found, the Inspection Team must decide if the product should be accepted, or revised and possibly re-inspected, or be completely re-written. It is through the decision that a group of people who have not been directly involved in tne development of programs signify that they are prepared to be regarded as co-responsible for the quality of the programs inspected.

Phase 4 - The Follow-up:
Management receives a report of the Inspection Team's decision, and statistics are collected of the number and types of errors found. Any re-work which needs to be done is the responsiblity of the Project Team, and their manager.

Two Problems in the use of Inspection Teams to control program quality

While tne Inspection Team Process has been used as a powerful application development tool, the use of this technique in the Standard Bank Data Processing Division created some non-trivial problems (see 53). Some of these problems are not relevant to the measuring and controlling of program quality, so they will not be discussed in detail - for example:-

- the role and training of Inspection Team Chairmen;
- the control of the Inspection environment to avoid the possibility of participants becoming involved in ego-problems;
- the cost of Inspections.

Two problems are relevant, and will be discussed in detail:-

Problem 1: The lack of consistency in the Inspection Team Process.

Although we went to great lengths to explain to staff members in Standard Bank Data Processing Division the objective of the Inspection Process and the definition of an 'error' (see Chapter 3), problems were experienced repeatedly in two areas related to subjectivity:-

i) Too often the Inspection Team wanted to suggest ways of correcting the errors which had been identified (which was already outside their terms of reference), and equally often there was disagreement among the Team members on which solution was most appropriate. This led to wasted time, and an attitude of scepticism among staff members concerning the efficacy of the process.

ii) Although a check-list of common errors was compiled from past Inspection reports to help both Project and Inspection teams to eliminate errors, the decision on whether a program should be accepted or revised (or worse, rewritten) was based on a subjective assessment.

It was not totally uncommon to hear programmers, whose work had not been passed by an Inspection Team, complain that they had been involved in a recent Inspection at which similar work had been regarded as acceptable.

A lack of consistency from the Inspection Teams not only leads to fustration among application development staff, it also demands that the value of Inspections as a quality control process be questioned. Perhaps, as it is presently constituted, the Inspection Team Process is not sufficiently objective to warrant regarding all programs passed by an Inspection Team as being of minimum standard quality.

Problem 2: The allocation of Inspection Teams on a Per Project Basis.

Because we felt that familiarity with the project would help Inspection Teams to work more effectively, at the Standard Bank we appointed an Inspection Team (and sometimes multiple teams) to a project for the duration of the project. Perhaps there is a danger in this approach. This point is raised in the Ethnotech Review Handbook (54), where it is suggested that perhaps an Inspection Team operating on this basis can become so involved in the project that it loses much objectivity. If this involvement becomes a problem, possibly it can be overcome by allocating the Inspection Teams on an ad hoc basis.

Some Benefits from the use of Inspections to control Program Quality

The benefits of Inspections are documented in the literature (see (55) and (56)). In the context of this research only those benefits which contribute to the control of program quality will be considered.

Benefit 1: An effective error identification technique.

It has been established that program quality is more than the absence of errors, however, it is inconceivable that a quality program will have errors in the code. The use of Inspection Teams is an effective way of identifying errors in a program.

i) Fagan's Claim (55)
Fagan claims that programs which have been inspected contain considerably fewer errors than programs which have been subjected to any other review technique. He bases this observation on a study of two pieces of operating system software. The process span during which the components were compared was 7 months beyond unit testing stage. The software which had been through the Inspection Team Process contained 38% fewer errors than the comparable piece of code which had not been inspected.

Because of the complexities of measuring programmer performance (see Chapter 5) there may well have been other factors which contributed to the number of errors found in each piece of code. However, the experience in the use of Inspection Teams at the Standard Bank also indicated that Inspections are an effective error-identification technique.

ii) Data from the Standard Bank
At the time that this information was extracted from the records of Inspection Team performance, the technique had been in use at the Standard Bank for 3 years.

To help substantiate the value of this process, the following data was compiled on the Inspections held during the previous seven months - (that is September 1978 to April 1979) It was found that:-

Number of Inspections held 154
Number of errors identified 864
(Since introducing the concept over 400 Inspections had been held and over 2200 errors identified).

Because it was felt that the raw information that 864 errors were found in 7 months had limited value, these errors were classified according to the definition of an error (see Chapter 3). The result of this classification is given in Table 9.

Table 9 Classification of errors according to the definition of an error.

Error Type	Number	Percent of Total
Logic Errors	449	52
Standards Violations	112	13
Specification Violation	147	17
Other e.g. Incorrect Comments	156	18
TOTAL:	864	100

For quality control purposes even this information was of limited value, so each of the 864 errors was again classified (Table 10). The purpose of the classification was to identify the top 10 errors made in the Department during that particular time-span.

Table 10 A further classification of errors.

Error	Percent of Total
Incorrect comments	14,5
Missing code	13
Standards violations	13
Redundant code	12
Incorrect logic	9
Incorrect initialization	8
Incorrect field definition	7
Technical errors	6,5
Misplaced full-stop	4
Incorrect data/paragraph name	4
Other (21 separate types)	9
	100

If it be accepted that Correctness and Reliability are attributes of program quality (see Section 7.4.2.2) then the error identification process of the Inspection Team helps towards controlling the quality of a programmer's output.