A Quality Assurance Reference Model For Object-Orientation

by

Deborah Thornton

555-853-0

Submitted in part fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

in the subject

INFORMATION SYSTEMS

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR : PROFESSOR A.L. STEENKAMP

June 1994

INDEX

ACKNOWLEDGEMENTS

Sincere and heartfelt thanks are due to Professor A.L. Steenkamp, my supervisor, for her invaluable advice, support and guidance throughout this project. Thanks also to Karen Richter, Bets Muller Amanda Storbeck and Theresa Wright for help with the layout and the drawings. To my husband, Julian and my son, Ashley, thanks are due for their indulgence as I neglected their needs in favour of my studies.

PREFACE

This dissertation has been done in partial fulfilment of the MSc degree in Information Systems at the University of South Africa.

A study of various quality assurance methods was made and the many problems surrounding software quality assurance reviewed. A Quality Assurance Reference Model is proposed taking object-orientation as the development paradigm and a specific software process model, namely the Revised Spiral Model as the point of departure.

The other degree requirements were the completion of five study modules, which were:

DATABASE DESIGN: This module covered requirement specification techniques, data models, design methodologies, database integration and scheme restructuring. It provides a background to the classic database design models.

EXPERT SYSTEMS: The concepts, characteristics and classification of expert systems were covered in this module. The architectures of different types of expert system were discussed and the manner in which expert systems are constructed was described.

SOFTWARE ENGINEERING: The software life cycle and software engineering environments were covered by this module. Also dealt with were requirements definition, software specification, software design approaches, programming considerations and the human aspects of software engineering.

OBJECT-ORIENTED ANALYSIS AND DEVELOPMENT: The object-oriented paradigm was analysed and pratical work done in terms of an object-oriented analysis and design.

INTELLIGENT DATABASES: Intelligent databases comprise a number of the latest techniques: object-orientation, expert-systems and hypermedia. All of these were discussed in some detail, as were database design methodolgies.

This study forms part of the Object Oriented Information Systems Engineering Environment Research Project (OISEE) currently under way at the University of South Africa. The aim of this project is the development of an information systems engineering environment within which team-sized research projects can be undertaken at postgraduate level.

The project addresses information systems engineering using the object-oriented paradigm throughout the development life-cycle.

ABSTRACT

The focus of the dissertation is on software quality assurance for object-oriented information systems development. A Quality Assurance Reference Model is proposed with aspects dealing with technical and managerial issues. A revised Spiral life cycle model is adopted as well as the Object Modelling Technique. The Quality Assurance Reference Model associates quality factors at various levels, quality criteria and metrics into a matrix framework that may be used to achieve quality assurance for all cycles of the Spiral Model.

KEYWORDS

Software Quality Assurance, Object-Orientation, Spiral Life Cycle Model, Quality Criteria, Quality Factors, Metrics, Quality Standards, Quality Management, Correctness Testing, Defect Prevention.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

CASE	Computer Aided Software Engineering
COCOMO	Constructive Cost Model
IS	Information System
KLOC	Thousand Lines of Code
LAN	Local Area Network
LOC	Lines of Code
OO	Object Oriented
OOA	Object Oriented Analysis
OOD	Object Oriented Design
OOP	Object Oriented Programming
QA	Quality Assurance
QCP	Quality Control Plan
SABS	South African Bureau of Standards
SQA	Software Quality Assurance
SQAP	Software Quality Assurance Plan
V&V	Verification and Validation

TERMINOLOGY

Attribute	a feature or property of an entity in which we are interested.
CASE	Computer Aided Software Engineering i.e. an integrated collection of tools designed to aid the life-cycle process of developing compliant software products.
Certification	the formal process of determining that a software product is suitable for an intended application.
Defect	a specific kind of unwanted condition or occurrence which has defied all attempts to be eliminated during development and so is delivered to the customer.
Development	all activities to be carried out in creating a software product
Error	an unwanted condition or occurrence which arises during a software process and deviates from its specified requirements.
Measure	an empirical objective assignment of a number (or symbol) to an entity to characterize a specific attribute.
Metric	an empirical objective assignment of a number or a symbol to an entity to characterize a specific attribute.
Model	an abstract representation of an object.
Repair	the process of restoring a non-conforming characteristic to a condition that the capability of an item to function reliably and safely, is unimpaired.
Software	an intellectual creation comprising programs, procedures, rules and any associated documentation pertaining to the operation of a data processing system
SQA	Software Quality Assurance is the degree to which a software product is in conformity with the established technical requirements.
SQAP	the Software Quality Assurance Plan is the plan contemplated to determine, or measure, the extent of software quality assurance.
Testing	the process of executing a program with the intent to yield measurable errors. There should be a means of determining whether or not the program has functioned as required.
Processes	any software related activities, normally having a time factor.
Products	any artefacts, deliverables or documents which arise out of processes.
Validation	the testing of software at the end of the development effort to ensure that it meets its requirements.
Verification	the evaluation of software during each life-cycle phase to ensure that it meets the requirements set forth in the previous phase.

TABLE OF CONTENTS

CHAPTER 1

THE CONTEXT OF QUALITY ASSURANCE

CHAPTER 2

SOFTWARE QUALITY

CHAPTER 3

THE OBJECT-ORIENTED DESIGN METHODOLOGY

CHAPTER 4

QUALITY ASSURANCE REFERENCE MODELS

CHAPTER 5

PROPOSED QUALITY ASSURANCE REFERENCE MODEL

CHAPTER 6

DEMONSTRATION OF CONCEPT

CHAPTER 1

THE CONTEXT OF QUALITY ASSURANCE

CHAPTER 1

THE CONTEXT OF QUALITY ASSURANCE

1.1 Introduction

As our dependence on computers in all spheres of life increases, the need for quality in software becomes more and more critical. Instead, as noted by most authors including Schach (1993) and Gillies (1992), software is often delivered behind schedule, over budget and with more errors than functionality. Added to this is the poor performance of the systems once they are finally delivered as well as the high cost of maintaining the systems, the lack of portability and the high sensitivity to changes in requirements. In 1968, at the NATO Software Engineering Conference it was claimed that building software is similar to other engineering tasks and that the philosophies and paradigms used by established engineering disciplines should be used in the development of software. Twenty-five years later, the so-called 'software crisis' is still with us, which would lead us to the conclusion that the software production process can not be tackled using established traditional engineering principles after all. However, the need for the establishment of methods ensuring that we produce quality software remains essential. If we cannot use existing methods from traditional engineering environments we must develop our own instead.

One must also be careful to distinguish between the concepts of 'quality assurance', 'quality control' and 'quality inspection'. Quality assurance is generally a guarantee with respect to certain software quality attribute measures falling within the prescribed performance standards on those quality attributes. According to Dunn (1990) :

"software quality assurance does not assure the quality of software, but the effectiveness of a quality assurance program."

Quality Control is the act of ensuring that the methods used to produce a product at all times conform to certain development standards, while Quality Inspection is the act of inspecting a delivered product to make sure that it meets some minimum defined set of standards.

Software quality assurance activities are generally related to those software verification and validation activities which are conducted throughout all stages of the software development lifecycle.

A set of reference models for Software Systems Engineering has been defined by du Plessis (1992). These reference models separate the concerns of particular aspects when designing software systems. One such aspect is concerned with quality assurance.

A quality assurance reference model should provide a frame of reference according to which the evaluation process for quality should proceed. This requires the identification of attributes to be measured, and metrics for the measurement of those attributes as well as metrics to determine the size and complexity of the proposed system. Software development standards should also be set up.

From a managerial perspective, it is necessary to identify a software development and testing process that will facilitate the production of high quality software. The model for this process is called the Software Development Process Model. Different methodologies will be investigated but object-oriented development will be our main focus because its cornerstones, namely data abstraction, inheritance and object identity are key features in the quest for quality.

In this chapter, the following issues are discussed:

- The relevance of quality assurance.
- The necessity for a methodology to control quality within software development.
- The research method used.
- Assumptions, hypotheses and constraints of the investigation.

1.2 The Relevance of Quality Assurance

The programming of non-trivial applications is a difficult and complicated affair. When a bridge collapses, it almost always means that the bridge must be redesigned and rebuilt from scratch. In contrast, when a software program fails it may be possible to simply restart the program and hope that the set of circumstances which caused the failure will not reoccur. Most software systems are designed under the assumption that all possible conditions cannot be anticipated so the software must be designed in such a way as to minimize the damage caused by an unanticipated condition. This highlights the difference between bridge building and software development: bridges are assumed to be perfectly engineered while software is assumed to be imperfectly engineered (Schach, 1993).

As Schach (1993) points out, a second major difference between bridges and software systems is in the area of maintenance. Maintaining a bridge is generally restricted to painting it and repairing minor cracks, resurfacing the road, and so on. A civil engineer, if asked to rotate a bridge through 90 degrees would consider the requestor to be bereft of his senses. It is not unreasonable, however, for up to half of a software program to be rewritten over a period of time. We think nothing of asking the software engineer to convert from batch operation to time-sharing, or to port a system from one machine architecture to another(figure 1.1). Often, this maintenance is undertaken by different people from those who produced the original software and the original documentation is often incomplete, incorrect or even non-existent. While an existing bridge is never half redesigned and rebuilt and, therefore, no incompatibilities between the old parts and the modifications and additions can arise, this is not the case when it comes to software. All too often, minor changes to a software product cause major problems when the system is recommissioned. While the civil engineer who designs a bridge is able to make technical drawings of his design, build prototypes and apply sound mathematical and engineering principles to it to see whether it is feasible, the software developer must often work to incomplete specifications and satisfy clients whose needs change at irregular intervals, and who have difficulty in communicating those needs to the developer.

Very often no measurable targets are set when developing software products. As Gilb points out (Gilb, 1988):

"Projects without clear goals will not achieve their goals clearly."

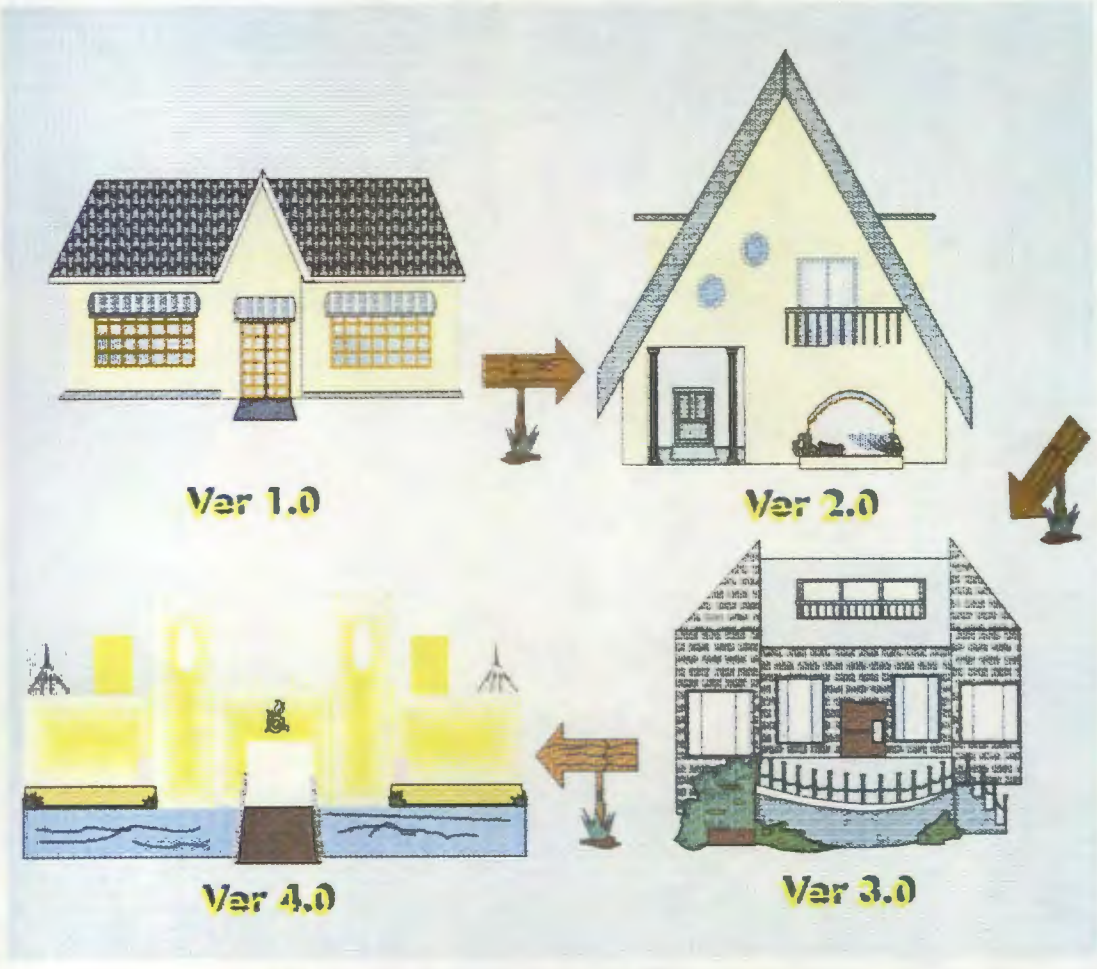

Fig. 1.1 **Software is often modified beyond recognition during its life cycle**

According to Fenton (Fenton, 1991), a major problem with the management of software development is the lack of rigourous measures and targets to determine what it is that is being attempted, and whether or not these goals have been achieved. He feels that managers not only fail to set measurable targets for software projects but they do not measure the various components which make up the real costs of software projects.

A further complication is the fact that software is becoming increasingly complex. In the seventies, software systems were mainly concerned with bread-and-butter systems such as payrolls. Nowadays, we expect software to be able to perform all sorts of complicated functions, very often in real-time and in a variety of distributed locations. Very complicated problems are addressed using computers, and these require in-depth understanding before it is possible to even begin to address solutions.

Because of the complexity of contemporary systems, whole project teams are set to

work on a single project and this 'programming-in-the-many' introduces further complications into the software development process. As Schach (1993) points out: "Team programming leads to interface problems between code components and communications problems between team members." Unless teams are properly organized, a lot of time can be wasted in conferences between team members.

New software and hardware for computers is released nowadays at a phenomenal rate with software packages being upgraded at least yearly. The capabilities of micro computers are increasing exponentially and multi-media systems are fast becoming the norm. For a novice, or even an experienced software engineer, just keeping up to date with this ever-changing technology is difficult. Whereas 10 years ago a software package was released with one small manual describing it functionality, nowadays most packages are accompanied by a whole series of manuals, not to mention on-line help. No sooner has the computer personnel managed to assimilate all the new information when a new version is announced.

One major area where quality assurance is mandatory is in safety critical software development. In systems where human lives might depend on the result of a computerised operation, there is no room for error. Poor quality software could not only inconvenience people, it could kill them. In Staffordshire, England, 1045 cancer patients were given the incorrect radiation doses due to a factor which was wrongly programmed into the computer. (Neumann,1994).

Once a software project is completed, it is very difficult to assess it. Does one rate its worth in terms of how well it conforms to the original specifications, or in terms of how satisfied the user is with it? Perhaps it should be evaluated with regard to how easy it is to maintain in later years, or by how few bugs are found in the system once it is in operation. The Department of Defence in the USA (1985) defines software quality as:

"the degree to which the attributes of the software enable it to perform its intended end use."

This definition can lead to problems as often, by the time a software product is delivered, its intended end-use has changed. Many other definitions of quality have been given over the years. Deciding which of these definitions is 'correct' with respect to software design is a major part of the problem when attempting to define quality in software.

As more money is allocated to computerisation, and as more and more operations are computerised, quality assurance techniques have become imperative to enable software developers to once again be able to produce working systems on time and within budget. Only when high quality software becomes the rule rather than the exception will the discipline of software development be accepted as a true engineering process. The human mind has a limited capacity and one of the reasons that this dissertation promotes object-orientation is the belief that this paradigm can effectively lift the ceiling of quality assurance.

1.3 Methodology for Software Development

Software design is a complex process during which the 'what' of requirements gets translated into the 'how' that leads to a coding solution. One important aspect of software design is a methodology. A methodology is a set of processes which steer the creation of the design in a certain direction. One type of methodology is the object-oriented approach adopted within the context of this investigation.

Object-orientation is a style which builds on previous structured techniques. A number of software engineering principles are formalised within this approach, specifically that of data abstraction. An abstract data type is a data type, together with the operations performed on instantiations of that type (Schach,1993). Information hiding consists of designing modules so as to localise to a single module a design decision that may be changed in the future. The progression of increasing abstraction culminates in a description of an object, namely an abstract data type that supports inheritance. According to Schach (1993), a design which uses an abstract data type is superior to one which merely uses a data structure together with the operations performed on that data structure.

Although object-oriented programming languages are useful in removing the restrictions which come about due to the inflexibility of traditional programming languages, it is the development process which has a major economic effect on the production of software. According to Schach (1993), about two-thirds of total software costs are devoted to maintenance. Software should, therefore, be developed under the assumption that it is going to be modified at some later stage. Modifications to one part of a system should not impact on any other parts of the system, if possible. The object-orientation characteristics of data encapsulation, inheritance and polymorphism go a long way to ensuring that such modularity in a system is possible to achieve and thus a system designed according to the principles of object-orientation should be far easier, and therefore cheaper, to modify and maintain. Schach also states that between 60% and 70% of all faults detected in large-scale projects are specification or design faults. Because object-orientation deals directly with objects which are real-world entities, object-oriented models facilitate problem identification, communication between the developer and the application expert, modelling enterprises and producing understandable documentation as well as program and database design. An object-oriented development approach encourages software developers to work and think in terms of the application domain through most of the software engineering life cycle.

According to Dunn (1990), the efficiency of languages addressing object-oriented programming is still in question, but as the greatest benefits of object-orientation come from helping specifiers, developers and customers express abstract concepts clearly and communicate them to each other (Rumbaugh et al, 1991) this should not be viewed as a major hurdle. From a programming viewpoint, object-orientation encourages reusability and portability as well as enforcing modularity, all of which have been identified as good software engineering principles.

Thus, it appears that object-orientation, by its very nature, if used correctly and fully,

Thus, it appears that object-orientation, by its very nature, if used correctly and fully, could go a long way to enabling us to produce high quality software systems. It is, therefore, this methodology that has been adopted for the purposes of this investigation and this paradigm is used throughout the software development life cycle.

1.4 Research Method

1.4.1 Scope of Research

The focus of this investigation is to establish a quality assurance reference model for software development. The achievement of high quality in a software system is a process, not an event, and one that concerns all parties to a software development project, including the users. Two aspects of quality assurance have been identified: the Technical Review Aspect and the Management Review Aspect.

1.4.1.1 Technical Review Aspect

This aspect deals with the definition and measurement of quality. Each major lifecycle step having a tangible outcome should have its quality tested as accurately as possible. In order to test quality, however, software metrics for quality assessment must first be identified. The methods of applying these metrics and analysing their results must also be defined.

1.4.1.2 Management Review Aspect

This aspect concentrates on the actual production of quality software. It is all very well to be able to identify quality but there must also be methods to ensure the making of quality software. Software quality assurance efforts should serve the needs of management in obtaining an efficient and effective software development venture and rapid solutions to difficulties at any phase of the software development process.

1.4.2 Assumptions and Hypotheses

As discussed in section 1.3, object-orientation has been decided upon due to the fact that object-oriented development has the potential to produce systems of higher quality than those produced by traditional means.

The software development process model for this investigation is the revised Spiral Model proposed by van der Walt and du Plessis (1994). This model was chosen firstly because it is believed that the Spiral Model alleviates many of the problems of traditional software development process models, and secondly because it has been revised specifically to cater for object-oriented development. The model is analysed in order to develop methods in support of both the Technical Review Aspect and the Management Review Aspect.

Before research can begin, certain hypotheses pertaining to the subject under investigation need to be made.

These include:

- the hypothesis that quality in software is, in fact, achievable by using a methodological approach, sound systems engineering principles and rigorous project management techniques.
- the analysis and design tools and methods, as well as the programming language used can influence the quality of a software project.
- the object-oriented methodology assures quality in software to a far greater extent than other methodologies.
- testing plays a very important role in assuring quality.
- the management of a software development project goes a long way to affecting the quality of the result.

1.4.3 Constraints

Because of the scope of the investigation, only a prototype quality reference model is to be designed. In order to arrive at this end, the following steps are followed:

- The state of the art of quality assurance expertise must first be examined and current trends and thoughts assessed. This will be done by a means of a in-depth look at some of the available literature.
- The results of the literature survey must then be analysed and relevant aspects identified.
- Once an exhaustive examination into the manner in which the problem has been tackled in the past has been achieved, an attempt will be made to conceptualize the problem, establish viable quality criteria and to synthesize one or more possible solutions for the future.

1.5 Demonstration of Concept

A scenario is presented within which the Quality Assurance Reference Model developed in this dissertation is applied to a small information system development project.

1.6 Dissertation Format

In Chapter 1 the need for a software quality assurance reference model was discussed in terms of the current software crisis in which software is rarely ever delivered on time, within budget, and without error. The object-oriented methodology and the revised spiral model for software development were proposed in terms of their applicability to alleviating the quality conundrum.

In Chapter 2 we will look at software quality assurance in terms of a software quality triangle (figure 1.2)

a. What is quality in software?
b. How do we measure it?
c. How do we achieve it?

Fig. 1.2 The Quality Triangle

This will be based on an in-depth survey of the available literature. The Technical and Management perspectives will be dealt with separately. Quality criteria will be identified and metrics and standards for quality assurance discussed. Management methods for enhancing productivity and quality will also be examined.

In Chapter 3 the object-oriented methodology will be investigated in greater depth and its relevance to quality assurance assessed.

Chapter 4 provides a high-level conceptualisation of existing quality assurance models while in chapter 5 these models are synthesized to produce a new Spiral Quality Assurance model specifically for the object-oriented methodology.

Finally, in chapter 6, the proposed model is demonstrated by means of a small prototypical example.

CHAPTER 2

SOFTWARE QUALITY

CHAPTER 2

SOFTWARE QUALITY

2.1 Introduction

'Software' and 'quality' are both intangibles. Yet in the last few decades software has come to play a key role in society. This means that some tangible solutions to aid with the achievement of the intangible called quality in the intangible called software are necessary.

We need, therefore, to define quality in such a way as to know how to go about achieving it, and to be able to identify it when it has been achieved. We also need to clearly identify technical and managerial ways of achieving quality. Technical people and their technology are key factors in software quality, while management, as the facilitator of the quality technologist and technology, also plays an important role.

2.2 The Technical Review Aspect of Software

Product quality is, at heart, a technical issue. According to Glass (1992):

"The creator of the software must strive for product quality, using the best technical tools and methods at his or her disposal, or quality will not happen."

As development proceeds, deliverables are reviewed and verified in terms of the requirements for completeness and consistency. There should be traceability from requirements through to implementation and the execution software should be validated with respect to the specified requirements. The roles of the various participants of the development process and the information resources should be defined and the evaluation attributes and metrics compiled. Adherence to established software engineering principles provide a point of departure for technical attributes

2.2.1 The Definition of Quality

Before a system's quality can be assessed it is necessary to know what exactly we mean by 'quality'. Unfortunately there doesn't appear to be one unambiguous definition available. Quality seems to be something that everybody has an intuitive feel for but which is especially hard to define, particularly in the area of software.

According to Macro and Buxton (1987)

"The two critical properties of a software system determining its quality, are its compliance with requirements as expressed in the functional specification and its modifiability for maintenance and new versions."

The reliability of the software will have bearing on both these issues and can be said to comprise aspects of both.

In his paper, Moretti (1990) defines high quality software as:
"correct, reliable and easy-to-modify software".
Trammel et al (1992) state that high quality software is:
"software that is correct, well documented, easy to read and understand, easy to maintain, and efficient."
Schach (1993) notes that although the word 'quality' implies excellence of some sort, the state of the art in software development is such that
"merely getting the software to function correctly is enough".
He therefore defines quality of software as:
"the extent to which the product satisfies all its specifications"
Gillies (1992) has given a few definitions of software quality from various sources:

a. Zero defects.
b. The totality of features and characteristics of a product or service that bear on its ability to satisfy specified or implied needs.
c. Fitness for purpose.
d. The degree of excellence.
e. The degree to which the attributes of software enable it to perform its intended end use.

Dr Edward Deming's definition of quality is:
"a predictable degree of uniformity and dependability at low cost and suited to the market."
The IEEE Standard for Software Quality Assurance Plans (1984) defines quality assurance as:
"a planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements."
In the IEEE Standard Glossary of Software Engineering Terminology, software quality is defined as:
"The degree to which software possesses a desired combination of attributes."
Finally, Schulmeyer et al (1987), after also listing a number of definitions of software quality taken from different sources, derive the following definition for software quality:
"The fitness for use of the total software product."

2.2.2 Why is Achieving Quality in Software so Difficult ?

As can be seen from the above, quality in software is very difficult to define and however we define it, there is still the problem of achieving this objective. Gillies (1992) points out that, firstly, quality is not absolute, it is multidimensional and is subject to constraints. He also notes that quality is about acceptable compromises and that quality criteria are not independent.

Thus, it appears that achieving quality in software is not only a difficult task, but it differs depending on the users' requirements. In fact, Peter Denning (1992) suggests that the question :

"What is software quality?"

be reframed as :

"How do we satisfy the customers of our software?".

He believes that by making the concerns of the customer central among the criteria for judging software, new actions will appear for making reliable and dependable software.

One reason for the difficulty in achieving quality in software lies in the fact that software is not visible in the way that a bridge or a house is. Only a model of the software can be conceptualized. Also, in most cases the user has a very incomplete understanding of his own needs at the start of development, and very often these needs change over time. However, most users have very high expectations with regard to the flexibility and adaptability of software. Added to this is the rapid rate of change in both software and hardware. Finally, very often there are tight time and cost constraints associated with any software development project.

An attempt to build a high degree of innate excellence into a software system is likely to be constrained by resources. Very often, the system that 'has it all' has proved unpopular with the users as being too complex, too expensive and has, in fact, adversely influenced productivity. Users do not like an interface which they consider to be too complicated, and very often do not appreciate getting more than they asked for as they find themselves getting lost in the added complexity.

One problem with building quality software is the gap in communication between the customer and the system developer as few customers have the expertise to read the formal specifications of a system. As it is these formal specifications and not the user specifications which form the basis of the contract between the user and the developer this can cause a severe problem.

Another problem is the fact that many different people handle the different aspects of the software lifecycle and the problem of too many perspectives comes into being.

Appropriate metrics or indicators of the degree of software quality are needed to obtain an early warning indicator of potential difficulties. This enables appropriate design changes to be made early in the software development life cycle where they are least likely to be problematic.

Two major techniques for testing quality assurance are verification and validation. Software quality assurance should lead to the detection of errors, and the diagnosis of these as either coding or logic errors. Coding errors are those introduced by the programmer who wrote the code and can be simple to locate, for example a plus sign instead of a minus sign in a calculation, or quite difficult to identify, for example, an incorrectly nested ENDIF statement.

Logic errors could be introduced at almost any stage during the development process and these errors are usually quite difficult to discover. Once they have been trapped, it should be ensured that the new logic is thoroughly verified before the accompanying code is written.

According to Sage et al (1990), software must be checked **structurally**, in terms of programming style and coding particulars, **functionally** to determine whether the software conforms to its specifications and **purposefully**, to determine if it actually does what the client wishes it to do. (figure 2.1)

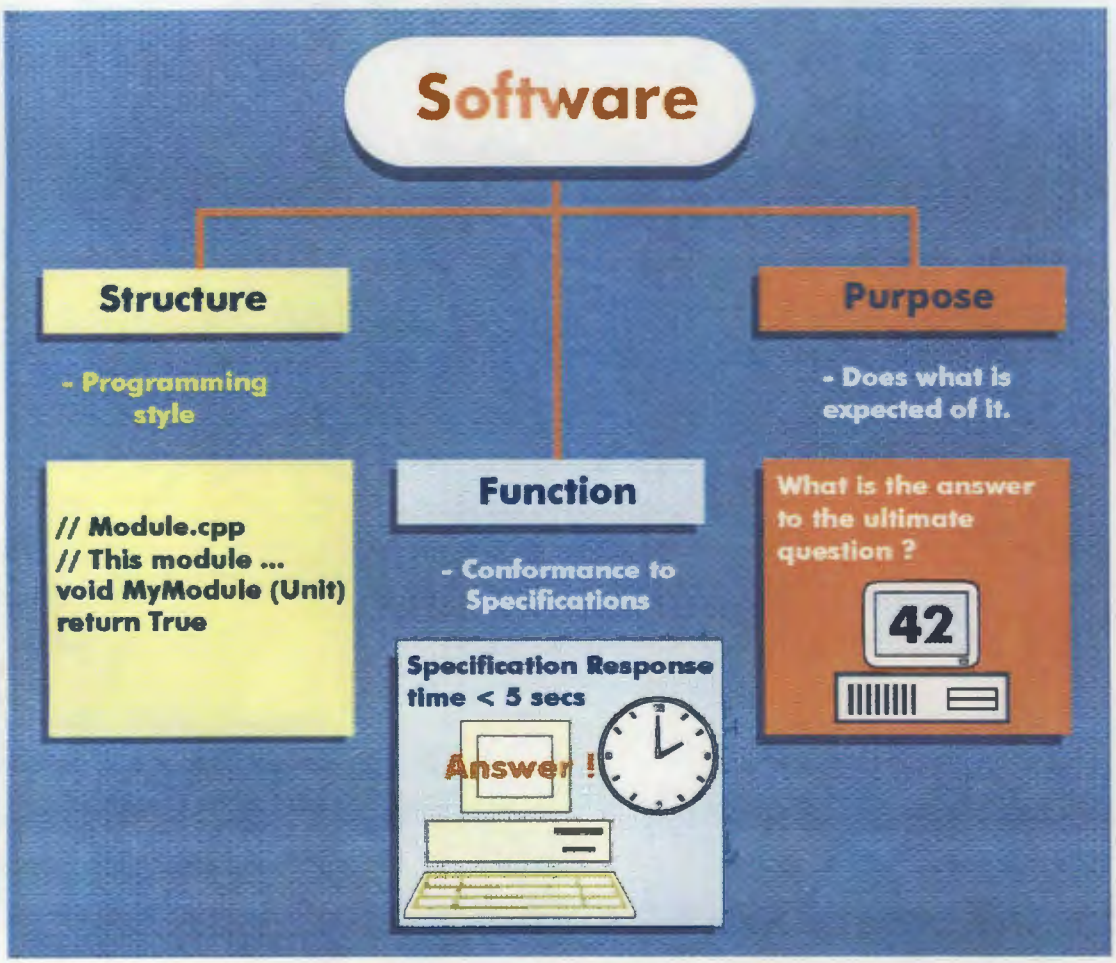

Fig. 2.1 **Aspects of Software to be checked**

2.2.3 Software Quality Measurement

2.2.3.1 Overview

According to Fenton (1991), measurement has been almost totally ignored within the mainstream software engineering. He believes that more often than not software engineers fail to set measurable targets when developing software products. He further suggests that software engineers do not measure the various components making up the real costs of software projects and that they do not attempt to quantify the quality of the products they produce.
There are various ways in which the quality of a software product can be measured. Firstly, the finished product can be tested with respect to certain software quality criteria (discussed in section 2.2.3.2). Secondly, special software measures, called metrics (discussed in section 2.2.3.5), can be applied to assess the quality of the system. Finally, laid down standards, whether these are national quality standards or simply internal company defined ones, can be applied to the system. Software metrics can also be used to estimate the size and cost of a system before any actual implementation takes place.

2.2.3.2 Software Quality Criteria

A high quality software product must score well on a number of criteria. These are mainly derived from non-functional requirements that ensure the operational functionality and trustworthiness of the software. Different authors identify different criteria as important, the following are a synthesis based on the ideas of numerous people including (Gillies, 1992; Denning, 1992; Dunn, 1990; and Glass, 1992). These criteria may be divided into two broad categories:
technical quality assessment criteria and user quality assessment criteria (figure 2.2).

Technical quality assessment would take into account the following aspects of a system :

a. Complexity — this includes the complexity of the algorithms used as well as the complexity of the system which is to be modelled.
b. Understandability — how easy it is to understand the system designer's intentions.
c. Maintainability — the ease with which the system can be understood, corrected, adapted and/or enhanced over its lifetime.
d. Reusability — the ability to make use of code or objects in other systems.
e. Efficiency — how well the system makes use of the available resources.

f. Testability	the ease with which as much as possible of the functionality of the system may be tested in a situation as close to real-life as possible.
f. Portability	the ability to move a system across operating environments.
g. Modifiability	the ability to change some of the functionality of a system.
h. Consistency	Whether the design techniques and coding methods used are consistently applied across the entire system. In large applications it is often necessary to provide extensive CASE tools to ensure that consistency is maintained.
i. Interoperability	Whether the system can communicate with other software packages e.g. import or export data or launch other programs.

User quality assessment would look at the following criteria :

a. User-friendliness	the ability of the system to be easily understood and used by human users.
b. Response Time	the response time of the system must be acceptable.
c. Reliability	whether or not the system can be relied on, firstly, to be available when needed, and secondly, to product correct information.
d. Robustness	the system should be able to withstand incorrect usage or conditions without crashing out every time.
e. Correctness	the extent to which a product satisfies its output specifications, independent of its use of computing resources, when operating under permitted conditions.
f. Integrity	avoidance of data corruption or loss.
g. Performance	the extent to which a product meets its constraints.
h. Security	avoidance of unauthorised access.
i. Flexibility	the ability of the system to satisfy different user requirements.
j. Completeness	the system must satisfy all the user requirements, not just a subset of them.
k. Utility	the extent to which a user's needs are met when a correct product is used under conditions permitted by its specifications.

Once the most relevant quality attributes have been identified for a particular system, importance weights for each attribute must be decided upon. Then it is necessary to determine operational methods for determining the scores for the attributes within specific software development approaches.

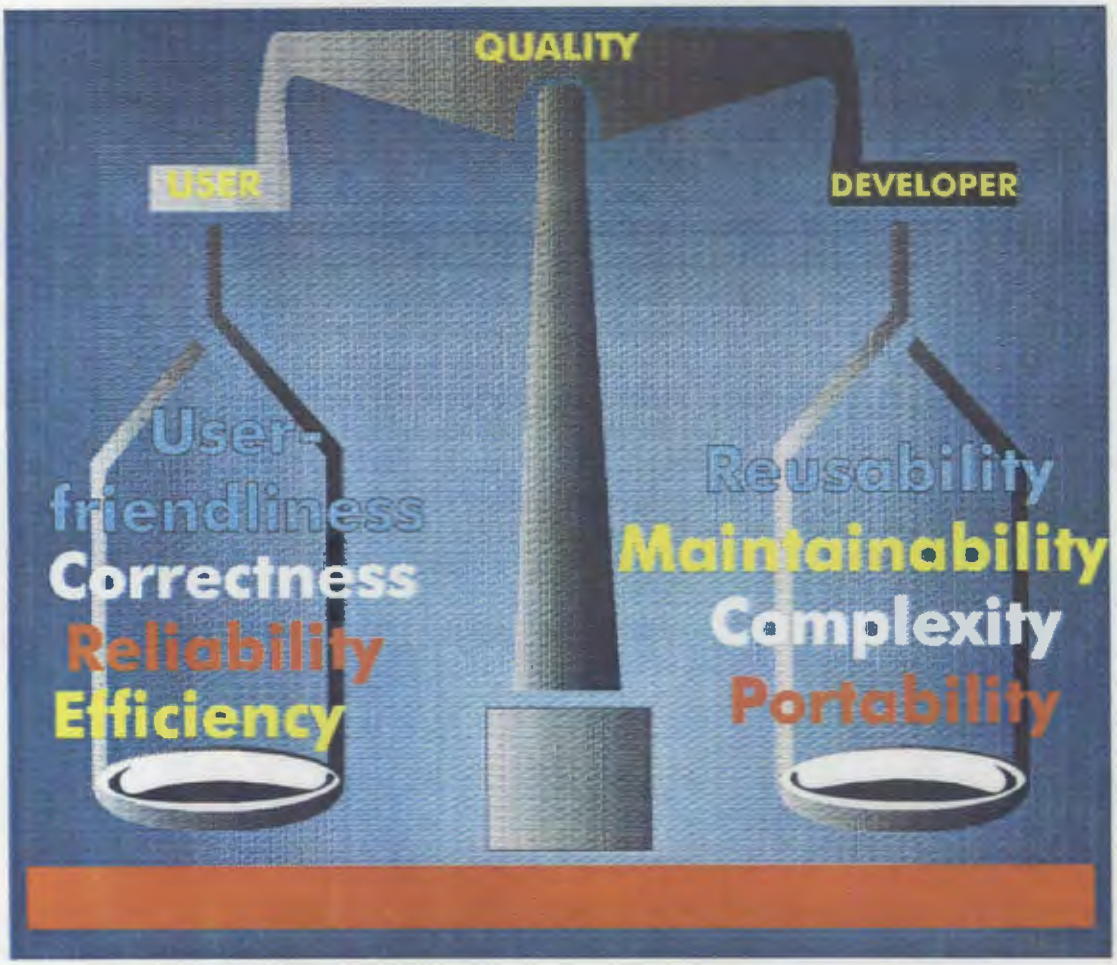

Fig. 2.2 The different viewpoints of User and Developer

There are a number of ways in which these criteria may be related to the quality assessment of the software.

Simple scoring	Each criterion is allocated a score. The overall quality is calculated by the mean of the individual scores. See table 2.1 for an example of this method.
Weighted scoring	The user assigns a weight to each criterion according to how important it is considered to be. Each criterion's score is then weighted before summation. This method is compared to simple scoring in table 2.1.
Phased Weighted Factor	A weighting is assigned to a group of characteristics before each individual weighting is considered. This approach is illustrated in table 2.2
Kepner-Tregoe Method	Criteria are divided into 'essential' and 'desirable'. A minimum value is specified for each essential criterion and software failing to reach these scores is considered unsuitable. 'Suitable' software is judged by use of the weighted factor method.

QUALITY CRITERION	METRIC	WEIGHT	PRODUCT
Usability	0.7	0.5	0.35
Security	0.6	0.2	0.12
Efficiency	0.4	0.2	0.08
Correctness	0.8	0.5	0.40
Reliability	0.6	0.4	0.24
Maintainability	0.6	0.4	0.24
Adaptability	0.7	0.1	0.07
Expandability	0.7	0.1	0.07
TOTAL	5.1	2.4	1.57

Simple Score = 5.1/8 = 0.64
Weighted Score = 1.57/2.4 = 0.65

Table 2.1 Example of Simple and Weighted Scoring
(Taken from Gillies (1992))

GROUP	CRITERION	METRIC	WEIGHT	PRODUCT	PWF
Product Opera-tion	Usability	0.7	0.5	0.35	
	Security	0.6	0.2	0.12	
	Efficiency	0.4	0.2	0.08	2/3
	Correctness	0.8	0.5	0.40	
	Reliability	0.6	0.4	0.24	
Product Operation Weighted Mean = 0.660					
Product Transi-tion	Maintain.	0.6	0.4	0.24	
	Adaptability	0.7	0.1	0.07	1/3
	Expand.	0.7	0.1	0.07	
Product Transition Weighted Mean = 0.633					
Overall measure by PWF method : ((2/3)*0.660) + ((1/3)*0.633) = 0.65					

Table 2.2 The phase weighting factor method
(Taken from Gillies (1992))

2.2.3.3 Software Quality Considerations

Interrelationships between Quality Criteria

It should be noted, however, that individual measures of quality may conflict with each other thus requiring compromises to be made. For example, the efficiency of software is usually inversely proportional to its integrity as error and integrity checking may be forfeited in the interests of speed. Maintainability too, may be adversely affected by improving the efficiency of a system. For example, a complicated algorithm might be preferred over a simpler one in the interest of the efficiency of the system, but the simpler algorithm might have been far more understandable to somebody who later has to maintain the system. Testability, portability, flexibility and reusability are also criteria which could be adversely affected by increasing the efficiency of the software.

This problem has been considered by Perry (1987), who has summarized the interrelationships between criteria as direct, neutral or inverse.

These relationships are summarized in figure 2.3

Correctness	C										
Reliability	Z	R									
Efficiency	Σ	Σ	E								
Integrity	Σ	Σ	Ω	I							
Usability	Z	Z	Ω	Z	U						
Maintainability	Z	Z	Ω	Σ	Z	M					
Testability	Z	Z	Ω	Σ	Z	Z	T				
Flexibility	Z	Z	Ω	Σ	Z	Z	Z	F			
Portability	Σ	Σ	Ω	Σ	Σ	Z	Z	Σ	P		
Reusability	Σ	Ω	Ω	Ω	Σ	Z	Z	Z	Z	R	
Interoperability	Σ	Σ	Ω	Ω	Σ	Σ	Σ	Σ	Z	Σ	I

Ω Inverse Z Neutral Σ Direct

Table 2.3 Perry's model of interrelationships

According to Dunn (1992), the most important criterion of quality is reliability. He defines software reliability as:

"the extent to which software can be depended on to correctly perform the functions assigned to it."

A reduction in the number of defects embedded in the fabric of the software is seen as the obvious avenue for improving reliability.

Narrow Focus of the Software Developer

It is a common problem that software developers tend to focus upon particular aspects of quality. When a user complains of poor quality, they tend to improve the product further in those areas. There are a number of ways in which quality of particular attributes can be enhanced. Glass (1992) maintains that the software developer must always be concerned with a certain minimum standard of quality, and in addition must employ special techniques to ensure successful achievement of the highest-ranked quality criteria. He proposes several techniques which he feels help achieve one or more of some of the above criteria.

Lack of Clarity

Another difficulty is that it is not always clear what is really meant by these software quality criteria. Taking completeness as an example, there are so many ways in which a software product can be incomplete that the development of a testable metric for such a criterion is not easy.

2.3.3.4 Software Quality Techniques

Single-point control

Single-point control is the notion of taking an action which must be done in several places, doing it once, and referencing that one occurrence from all the places where it is needed.

High-Level Language

Quality is usually enhanced by using the highest-level language that solves the problem at hand. However, it may be necessary to move down the language ladder until a language which is efficient enough is found as efficiency is usually negatively correlated with the order of language.

Reviews

The review is a fundamental quality technique. Types of reviews are requirements reviews, design reviews, peer code reviews, change reviews, test reviews and post-delivery reviews.

Standardization

A useful way to achieve portability is to use a language, computer and operating system which are common. For example, a programming language such as C runs on a portable operating system and utilizes a portable compiler.

Design for Efficiency

If the effort to achieve efficiency within a program is postponed to the coding phase, Glass (1992) feels that it may well be too late to achieve the necessary level of this criterion. The operation computers do least efficiently is accessing storage devices so it makes sense to design for a minimum level of input/output activities.

Think User

According to Glass (1992), it is no longer enough to build software that does its job well - it must also be convenient and comfortable to use. He believes that the key to good user interface design is to think like a user. This can take on many dimensions:

- Market research — Study what the user wants as well as what he needs.
- User participation — Let your user give feedback as the design evolves.
- Prototyping — Build a sample for the user to experiment with.
- User education — Offer courses in product use.
- Levels of support — Provide a tutorial interface and optional access to help files. Produce meaningful diagnostic messages.

Readable Code

Use naming conventions, indentation and comments to make code more readable.

2.2.3.5 Software Metrics

Proponents of software quality measurement have said that we cannot control what we cannot measure. Fenton (1991) believes that this argument is incomplete in that often the specific motivations for measuring something are either unclear or do not even exist. He outlines the kinds of things he feels need to be measured in order to meet various objectives. He feels that firstly, managers need to do the following :

1. Measure the cost of the various processes within software production.
2. Measure the productivity of staff in order to determine pay settlements.
3. Measure the quality of the software products produced in order to compare different projects, make predictions about future ones, and establish base lines and set reasonable targets for improvement.
4. Define measurable targets for projects.
5. Measure repeatedly particular process and resource attributes in order to determine which factors affect cost and productivity.
6. Evaluate the efficacy of various software engineering methods and tools, in order to determine whether it would be useful to introduce them to the company.

Engineers, on the other hand, need to do the following :

1. Monitor the quality of evolving systems by making process measurements.
2. Specify quality and performance requirements in strictly measurable terms, in order that such requirements are testable.
3. Measure product and process attributes for the purpose of certification.
4. Measure attributes of existing products and current processes to make predictions about future ones.

Quality measurement is usually expressed in terms of metrics. A software metric is a measurable property which is an indicator of one or more of the quality criteria that we are seeking to measure (Gillies, 1992). The purpose of software quality metrics is to make assessments throughout the software life cycle as to whether the software quality requirements are being met (Schulmeyer et al, 1987). The use of metrics reduces subjectivity in the assessment of software quality by providing a quantitative basis for making decisions about software quality. However, the world is made up of things which are not easily measured. There are times when a good dose of intuitive decision making based on the input from an experienced person is better than any amount of questionably generated numbers could ever be. Thus, the use of metrics does not eliminate the need for human judgement in software evaluations.
The key to a successful quality assurance group (refer to section 2.6.3.1) is to define and measure quality. Software metrics measure various attributes of software and relate to different aspects of software quality. They provide a more disciplined, engineering approach to quality assurance and a mechanism allowing a life-cycle viewpoint of software quality.
The actual measurement of software quality is accomplished by applying these metrics to the documentation and code produced during software development. Metrics may be classified into three categories:

1. Anomaly detecting	identify deficiencies in documentation or source code e.g. standards enforcement.
2. Predictive	measurements of the logic of the design and implementation. They provide an indication of the quality that will be achieved in the end product.
3. Acceptance	measurements that are applied to the end product to assess its final compliance with requirements.

There are a number of criteria which a quality metric must meet, namely :
- a metric must be clearly linked to the quality criterion that it seeks to measure.
- it must be sensitive to the different degrees of the criterion.
- it must provide objective determination of the criterion that can be mapped onto a suitable scale (Gillies, 1992).

A metric is a qualitative indicator, not an absolute measure. Software metrics can be divided into two types: predictive or descriptive. Predictive metrics make predictions about the software later in the life cycle while descriptive metrics describe the state of the software at the time of measurement.

Schulmeyer et al (1987) also divide metrics into two types based on levels of validation. A metric is said to be internally valid if, in the sense of measurement theory, it provides a numerical characterization of some intuitively understood attributes. It is said to be externally valid if it can be shown to be an important component or predictor of some behavioural software quality attribute.
Gillies (1992), give us several criteria that make a good metric:

a. Objectivity	The results must be free from subjective influences.
b. Reliability	The results should be precise and repeatable.
c. Validity	The metric must measure the correct characteristic.
d. Standardization	The metric must be unambiguous and allow for comparison.
e. Comparability	The metric must be comparable with other measures of the same criterion.
f. Economy	The simpler the metric is to use, the better.
g. Usefulness	The measure must address a need and not simply measure a property for its own sake.
h. Consistency	A metric should be dimensionally consistent.
i. Automation	A metric that lends itself to automation is desirable since it will be objective and boost both economy and convenience.

Seven measurable properties can be identified upon which most metrics appear to depend :

Readability

The readability of the documentation allows one to assess how useful such documentation may be in the enhancement of the usability of a piece of software. Two specific metrics have been proposed : The Flesch-Kincaid Readability Index and the Fog Index.
Readability has also been suggested as a measure of maintainability.

Error Prediction

Ottenstein uses some basic parameters such as the number of operators and operandi in a program to predict the number of errors to be found. This provides a measure of the correctness of the system.

Error Detection

The Remus and Zilles model of defect removal uses the number of detected errors and a measure of the error detection efficiency to predict the number of errors remaining undetected in the system. In other words, this can also be used to determine the correctness of the system.

Mean Time to Failure

This may be assessed by measurement, estimation or prediction and is a measure of the reliability of the system.

Complexity

Based on the assumption that as complexity increases so reliability decreases, measures of complexity provide an indication of the reliability of the system. A number of different ways of measuring complexity have been proposed, including McCabe's Cyclomatic method, knot calculations and measures devised from Halstead's "software science". McCabe's measure derives directly from graph theory and is easily computed once the graph of the program has been generated.

Knot counts measure the number of excursions from sequential execution of processing nodes.

Halstead's "software science" uses simple computation to combine the number of unique operators, unique operands and the repetition in using these to compute a number called the program volume.

Complexity is also indicative of the maintainability of the system.

Modularity

Increased modularity is generally assumed to lead to increased maintainability. There are currently four different ways in which modularity may be measured.

Testability

The ease and effectiveness of testing will have an impact in the maintainability of the product.

Reliability

Keene states that there are at least three methods for estimating what is perhaps the most important of the quality criteria: reliability. These methods are based on:

1. Precedent Developed Code
2. The Fault Content Model
3. The Time Domain Model

However, as can be seen from the above, these measures, and the metrics they represent, relate to only a few quality criteria and this in a complex many-to-many type of relationship. Gillies (1992) maintains that of the 40 metrics identified by Watts, 18 of them measure maintainability, 12 measure reliability, 4 measure usability and 3 measure correctness.

Integrity, expandability and portability are each measured by a single metric and no metrics exist with which to measure efficiency, adaptability, interoperability and reusability.

Thus, Gillies (1992) identifies the following problems with metrics:

a. They cannot be validated.
b. They are not generally objective.
c. Quality is a relative, not an absolute, quantity.
d. Metrics depend on a small set of measurable properties.
e. They do not measure the complete set of quality criteria.
f. Many metrics measure more than one criterion.

Dunn (1990) defines some less formal quality measures which are also extremely valuable. These measures have the advantage of being easily measurable, without the need for formal measuring techniques. They include:

a. Failures per unit of time under operational circumstances.
b. Operational 'incidents' per unit of operating time.
c. Calls for assistance per month.
d. Number of known defects (whether or not normalized to LOC).
e. Number of undiagnosed test failures (whether or not normalized to LOC).
f. Ratio of tests passed to tests run.

Glass (1992) mentions a proposal of the "Ten Best Data Processing Measures" which were identified at the Third National Conference on Measuring Data Processing Quality and Productivity held in 1985. These measures are:

1. Defects per 1000 lines of code.
2. Mean time between failure.
3. Return on investments.
4. Spoilage (cost of rework/development cost)
5. Change from baseline data.
6. Productivity increase.
7. Degree of risk in application.
8. Cost per unit of work.
9. Over/under measures (actual vs projected)
10. User satisfaction.

2.2.3.6 Quality metrics for object-oriented development

Moreau (Moreau,1989) feels that traditional metrics are inappropriate for object-oriented systems for several reasons:

1. The assumptions relating program size and programmer productivity in structured systems do not apply directly to OO systems.
2. Traditional metrics do not address the structural aspects of OO systems.

3. The computation of the system's complexity as the sum of the complexity of the components is not appropriate for OO systems.

Tegarden et al (1992) challenges this view. They believe that there are valid reasons for evaluating the use of traditional metrics for OO systems:
1. The traditional metrics exist.
2. There is empirical evidence to support their use for structured systems.
3. They are well understood by practitioners and researchers.

Software complexity is an area of software engineering concerned with the measurement of factors that affect the cost of developing and maintaining software. Traditional metrics have been applied to the measurement of software complexity for many years. In OO systems, complexity is reduced by means of polymorphism and inheritance. Polymorphism means that a programmer does not have to comprehend, or even be aware of, existing operations in order to add a new operation. Also, the programmer does not have to consider methods that are not part of the object when naming the operation. Inheritance decreases complexity in programs because it enables programmers to reuse previously defined objects, thus reducing the number of operations and operands required. Laranjeira (1990) proposes an object-oriented model for functional specification which reflects these characteristics. He states that a specification effort should begin by identifying the entities in a problem domain and their interrelationships, and continue by detailing the functions performed by, as well as the internal state of each object. This conforms to the object and dynamic models in the Object Modelling Technique as described by Rumbaugh (1991).
Objects which allow partitioning and the layers of abstraction in each partition could be identified. A major advantage of using objects is that a direct and natural correspondence with the real world is possible. Problem domain entities are extracted directly into the model without any intermediate buffer such as traditional data flow diagrams being necessary (although data flow diagrams are used in the functional model of the Object Modeling Technique, they are not the traditional DFDs). This leads us to the belief that an object-oriented representation of a system is a more suitable model for accurate software size estimates than one achieved through a more traditional approach.
Given the relative newness of the object-oriented paradigm for software design, metrics specifically disposed towards object-orientation are needed as software metrics developed specifically for traditional methods do not consider such notions as classes, inheritance, encapsulation and message passing. Chidamber et al (1991) believe that specific metrics designed for the object-oriented approach can be very useful in evaluating the degree of object-orientation of an implementation as well as providing objective criteria in setting design standards for an organisation.

Laranjeira (1990) summarizes the size estimation process into the following steps:

1. Beginning with the lowest level of decomposition, evaluate the size of each object.
2. Continue to higher levels.
3. Include 'utility objects' to account for housekeeping functions, if necessary.
4. The estimated size of the system will be the sum of the size estimates of the objects in the top level decomposition plus the size of any 'utility objects'.

Nonfunctional specifications such as memory usage, speed or time constraints should also be taken into account as they might also influence software size. Chidamber, Shyam and Kemerer (1991) present six candidate metrics geared for object-orientation in their paper. A summary of these is given below :

a. Weighted Methods Per Class (WMC)
WMC is equal to the sum of the static complexities of all methods in a class.

b. Depth of Inheritance Tree (DIT)
DIT is the measure of how many ancestor classes can potentially affect this class.

c. Number of Children (NOC)
NOC is a measure of how many sub-classes are going to inherit the methods of a parent class.

d. Coupling Between Objects (CBO)
The CBO metric for a class is a count of the number of non-inheritance related couples with other classes.

e. Response for a Class (RFC)
RFC is equal to the cardinality of the response set of the class.

f. Lack of Cohesion in Methods (LCOM)
LCOM is equal to the number of disjoint sets formed by the intersection of the sets of all instance variables used by each of the methods in a class.

2.2.3.7 Software Quality Standards

One of the underlying problems associated with acquiring quality software is the lack of visibility and control over the software development process. One predominant cause of this lack of visibility is the lack of quality standards. Standards are important in the regulation of the production of software and its deployment. They may be used to ensure:

- that individual items of software are fit for the purposes intended.
- the independent operation of pieces of equipment.
- a free market for third party suppliers.
- the regulation of the development of software within a company.

The ISO9001 standards lay out the requirements for a generic quality system for two-party contractual situations. Because the process of development and maintenance of software is different from that of most other types of industrial products, additional guidance for these types of systems are provided in the ISO9000-3. Cross-reference indexes, in the form of annexures to the ISO900-3, are provided.

A summary of the sections outlined in the ISO9001 follows:

Clause 4.1: Management Responsibility.
The model emphasizes the importance of management in quality control throughout the organisation. The clause sets out the basic principles for establishing the quality system within the organisation and sets out many of its functions, which are described in detail in later sections.

Clause 4.2: Quality System
The model requires the organisation to set up a quality system. The focus of the plan should be to ensure that activities are carried out systematically and that they are well documented.

Clause 4.3: Contract Review
This specifies that each customer order should be regarded as a contract. Customer requirements should be clearly defined and in writing. Differences between the order and the original quotation should be highlighted. It should be ensured that the requirements can, in fact, be met.

Clause 4.4: Design Control
Design control procedures are required to control and verify design activities, to take the results from market research through to practical designs.

Clause 4.5: Document Control
Three levels of documentation are recognised by the standard.

Clause 4.6: Purchasing
The purchasing system is designed to ensure that all purchased products and services conform to the requirements and standards of the organisation. The emphasis should be placed on verifying the supplier's own quality management procedures.

Clause 4.7: Purchaser supplied product
All services and products supplied by the customer must be checked for suitability.

Clause 4.8: Product identification and traceability
Procedures must be established to identify and trace materials from input to output.

Clause 4.9: Process Control
This must be documented and procedures for setting up or calibration must also be recorded.

Clause 4.10: Inspection and Testing
This is required to ensure conformance on incoming materials and services, 'in process' to ensure that all is going according to plan, and on the finished product or service.

Clause 4.11: Inspection, measuring and testing equipment
Any equipment used for measuring and testing must be calibrated and maintained.

Clause 4.12: Inspection and testing status
Materials and services are either awaiting inspection or testing, or they have either passed or failed inspection. This status should be clearly identifiable at any stage.

Clause 4.13: Control of non-conforming product
Although this clause is not prescriptive about performance levels, all non-conforming products or services need to be clearly identified and documented. Procedures to handle these products should be established.

Clause 4.14: Corrective action
Corrective action should be implemented via a systematic programme and records should be kept of any action taken.

Clause 4.15: Handling, storage, packaging and delivery
This clause covers all activities which are the contractual obligation of the supplier with regard to the handling of the product.

Clause 4.16: Quality records
These form the basis for quality audits. Existing practice should be assimilated wherever possible in order to reduce rework in the reproduction of previously established quality records.

Clause 4.17: Internal quality audits
The quality system should be inspected from within the organisation according to established procedures. Internal audits should be carried out in order to identify problems early on in the development cycle.

Clause 4.18: Training
Written procedures should be produced in order to establish training needs, carry out effective training and to record the training requirements and completed activities of all personnel.

Clause 4.19: Service
Documented procedures should exist to ensure that servicing is actually carried out and that there are sufficient resources available to provide this facility.

Clause 4.20: Statistical Techniques
The standard does not specify particular techniques or methods but says that those used should be appropriate for the intended purpose.

All a standard does, however, is to establish the model to be employed. An accreditation body (the South African Bureau of Standards in South Africa) must then be called upon to ensure that the implementation meets the required standard and continues to do so over time.
In order to gain accreditation, an organisation should implement a quality system in accordance with the requirements of the standard. The accreditation body will require a pre-inspection examination of the relevant documentation. They will then visit the organisation to ensure that the system meets the required standard. The certification of accredation may be withdrawn at any time if the system is not properly maintained.
The standard is based on a model which employs two fundamental principles: Right First Time and Fitness for Purpose. It is intended to be realistic and implementable and sets no prescriptive quality targets, referring instead to standards agreed as part of the contract with the customer and acceptable to them. The standard focuses on ensuring that procedures are carried out systematically and that the results are documented.
Although he feels that ISO9001 can be an extremely powerful incentive to organizations to get their quality procedures right, Gillies (1992) feels that
"There is very little in the standard about establishing a human quality culture."

This is a very important observation. Until quality control becomes a part of the culture of the people involved, standards merely force the supplier to establish certain procedures in a prescriptive manner instead of simply existing as a means of checking that the people involved are doing their work properly. As Gillies (1992) observes,

"Without the establishment of a quality culture and a formal requirement for procedures to facilitate this process, the vital process of continuous improvement which takes quality management beyond the recording of errors and performance may be omitted."

One of the biggest barriers to the acceptance of ISO9001 in the field of software is the fact that it originated as a manufacturing standard. Thus, ISO9000-3 was published in 1991 in order to explain how the standard should be applied in a software context.

ISO9000-3 recommendations comprise of four parts:

1. Introductory Material

The first 3 clauses of the standard define its scope, references to other standards and certain terminology used.

2. Section 4: Quality System - framework

This provides comprehensive definitions as to the responsibilities of the various parties involved in the software development process. The management responsibility of the supplier is laid out in the following Quality Policy taken from SABSISO 9000-3,

"The Supplier's management shall define and document its policy and objectives for, and commitment to, quality. The supplier shall ensure that this policy is understood, implemented and maintained at all levels in the organisation."

The standard also prescribes the precise definition of roles of all personnel who manage, perform or verify work affecting quality. In-house verification requirements should be identified and the relevant resources for this be provided by the supplier of the software. Regular joint reviews between supplier and purchaser are recommended. Other recommendations include:

A documented 'quality system' should be established and maintained by the supplier. This should be an integrated process to ensure that quality is built into the system as development progresses.

Internal quality system audits should be scheduled to determine the effectiveness of the quality system. Procedures for correctives actions should be established, documented and maintained by the supplier.

3. Section 5: Quality System - Life-Cycle

Activities ISO-9000 is intended to be used irrespective of the life-cycle model used, but it is prescribed that a software development project should be organised according to a life-cycle model. Recommendations pertaining to contract review, requirements specification, development planning, quality planning, design and implementation, testing and validation, acceptance testing, delivery and installation and maintenance are given.

4. Section 6: Quality System - Supporting Activities

In this section such topics as configuration management, document control, quality records, product and process measurement, rules, practices and conventions, tools and techniques, purchasing, included software product and training are discussed.

Software is considered different from other applications because :

- it is considered as an intellectual object
- the development process has its own characteristics and importance
- replication always gives an exact copy
- software does not wear out
- once a fault is fixed it will not re-occur.

Thus, the ISO9000-3 is not a new or different standard. Quality systems are still assessed against ISO9001 with ISO9000-3 provide guidance *"where a contract between two parties requires the demonstration of a supplier's capability to develop, supply and maintain software products"*.

Although the above international standards are important and very valuable, Glass (1992) asserts that the present state of the practice is that most standards are homegrown. He also maintains that there are two major problems with the use of standards in current practice. The first problem is that most computing installations have too many standards and the second is that most computing installations do little to enforce these standards. He alleges that standards should be short and to the point and should be distilled into a short manual which can be digested quickly, applied easily, and enforced conveniently. More elaborate, preferred but not required ways of building software should be specified as guidelines to software development rather than standards.

2.2.3.8 Software Quality Indicators

Software quality indicators are not to be confused with software quality metrics. Software quality indicators provide trends rather than absolutes. They do not, nor are they intended to, replace sound quality practices. If they are properly applied and meticulously followed up, they will indicate those areas requiring additional management attention. Schulmeyer et al (1987) name seven software quality indicators: completeness, design structure, defect density, fault density, test coverage, test sufficiency and documentation.

The completeness indicator should be used throughout the entire software development lifecycle. It provides insight into the adequacy of the software specifications and can be used to identify specific problem areas within the software specifications and design.

The design structure indicator looks at the clarity of the design, independent of the complexity of the implemented functions.

The defect density indicator provides early insight into the quality of the software design and implementation into code.If, after a design and code inspection, the defect density is outside of the norm for a particular software development effort, it is an indication that the development and/or inspection process may require further scrutiny.

The fault density indicator is very similar to the defect density indicator. The primary differences are the application phases and an emphasis on test instead of inspection data. This indicator can be used in conjunction with the test coverage indicator to assess the software reliability and maturity. It can also be used to determine if sufficient software testing has been accomplished.

The test coverage indicator presents a measure of the completeness of the testing progress from both a developer and a user perspective.

The test sufficiency indicator is a useful tool in assessing the sufficiency of software integration and system testing, based on the prediction of the remaining software faults. This prediction is, in turn, based on the experienced fault density.

The primary objective of the documentation indicator is to gain insight into the sufficiency and adequacy of the software documentation products that are necessary in the operational and post deployment software support environments. It also provides a mechanism for the identification of potential problems in the deliverable software documentation and source listings that may affect the usability and maintainability of the operational and support software.

2.2.3.9 Paradigms and Languages

Programming Language

Apart from testing, the last stage of programming is the final encoding of the solution. Whatever the problem, the results of the design process must be conveyed to the computer. The most basic method is by means of assembler language. Dunn (1990), however, feels that :

"Assembler language is tedious to write, difficult to read, and exposes the programmer to a level of hazard hidden by other languages. It is excusable only for time-critical parts of certain real-time programs."

Assembler language should, therefore, be ruled out as a mechanism for quality programming. According to Glass (1992):

"In general, quality is achieved by using the highest-level language consistent with the problem to be solved."

The reasons for choosing a high-level language over a lower level one are many:

- High-level languages finesse whole levels of solution effort, eliminating chances for error.
- There are fewer lines of code in a high-level solution, and thus fewer chances for error.
- Structured code is facilitated by high-level languages.
- Portability, maintainability and testability are facilitated by higher-level languages.

Thus, it appears that most quality criteria are enhanced by the use of high-level languages, efficiency being one exception. Problems which need an efficient solution will find themselves moving down the language-level chain until the best compromise between the quality attributes can be obtained.

Most software written today is written in languages which deal with the data and the procedures that use the data separately. The exception to this is object-oriented programming. Object-oriented programming is based on the concepts of objects and messages.

Information Hiding, cohesion and coupling

Whatever type of language used the goal is the same:

"Replace the complexity of the programming problem with a set of inherently less complex constituents of a programming solution." (Dunn, 1990).

Modularity is the attribute that describes the mutual independence of the elements of the solution. Greater independence ensures greater likelihood that individual design efforts will result in a working and maintainable system. Modular programming is the practice of implementing software in small, functionally oriented pieces. Each module is devoted to one or more tasks related to a function and may be accessed from one or several places in a

software system. By isolating functions into separate code units, several advantages are gained. According to Glass (1992), the software is more easily designed, built comprehended, tested and modified, since the structure is easily related to the tasks being performed.

One of the best known methods for reducing the programming problem is based on the concept, proposed by David Parnas, of information hiding. The idea is that the specification of any piece of the solution should contain nothing beyond the minimum amount of information required to implement the specification. One formal method for achieving information hiding is known as structured design. Structured design is concerned with the coupling among modules which makes them interdependent and the self-sufficiency of modules, or cohesion, that promotes their independence.

Reusability

Reusing previously designed and tested code sounds like a marvellous idea, however there is a cost associated with finding existing components that can fit into a software scheme. Reusability can apply at any level of component hierarchy and can be applied to design as well as code. According to Dunn (1990), most examples of successful reuse arise from staff members' recognition that the problem currently being solved has elements in common with one or more systems developed earlier. However, even if the programmer is the same one who solved the previous problem, he or she may still prefer to redo the component to perfect it. More likely, it is not the same programmer who is working on the new problem and most programmers prefer to start afresh than to take time to understand somebody else's code well enough to modify it. Also, programmers seldom have any means of determining if existing code, other than their own, is a direct or near fit to the current problem.

2.3 The Managerial Review Aspect of Software Quality

2.3.1 Introduction

Quality is fundamentally a management problem. Gillies (1992) feels that without management commitment and belief, any initiatives to improve quality are doomed. Management shows its commitment to quality by establishing a quality policy which should set the expectations for quality and should apply to all departments and individuals within the organisation. Software engineering was introduced to try to formalise the development of software using ideas from other engineering disciplines. The development of systematic procedures to produce structured code, which became known as methodologies, was the first

widespread attempt to take account of quality issues during software development.

According to Thayer et al (1980), there are twenty problems in software engineering project management which urgently require solutions in the drive to produce quality software. These twenty problems are :

1. Requirements specifications are frequently incomplete, ambiguous, inconsistent and/or unmeasurable.
2. Success criteria for a software development are frequently inappropriate resulting in poor-quality delivered software.
3. Planning for software engineering projects is generally poor.
4. The ability to accurately estimate the resources required to accomplish a software development project is mediocre.
5. The ability to accurately estimate the delivery time for a software development project is meagre.
6. Decision rules for use in the selection of the correct software design techniques, equipment and aids to be used in the design of the software are not usually available.
7. Decision rules for selecting the correct procedures, techniques, strategies and tools to be used when testing software are not available.
8. Procedures, techniques and strategies for designing maintainable software are not always available.
9. Methods to guarantee that the delivered software will work for the user are not available.
10. Procedures, methods and techniques for designing a project control system that will enable project managers to successfully control their projects are not readily available.
11. Decision rules for selecting the proper organizational structure are not available.
12. The accountability structure in many software engineering projects is poor, leaving some question as to who is responsible for various project functions.
13. Procedures and techniques for the selection of project managers are often ill-advised.
14. Decision rules for use in selecting the correct management techniques for software engineering project management are not available.
15. Procedures, techniques, strategies and aids that will provide visibility of progress to the project manager are not available.
16. Measurements of reliability that can be used as an element of software design are often not available and there is no way in which to predict software failure.

17. There is no practical way to show that one program is more maintainable than another.
18. Indexes of 'goodness' of code that can be used as an element of software design are not available.
19. Standards and techniques for measuring the quality of performance and the quantity of production expected from programmers and data processing analysts are not available.
20. Techniques and aids that provide an acceptable means of tracing a software development from requirements to completed code are not generally available.

Not all of the above problems are managerial, some are technical but they are mentioned together under the managerial aspect for continuity and because the technical problems cannot be solved without management support.

2.3.2 Quality Management Systems

2.3.2.1 Overview

The ISO (ISO8042, 1986) defines a quality management system as
"The organizational structure, responsibilities, procedures, processes and resources for implementing quality management."
The quality management system (QMS) should provide a structure to ensure that the process is carried out in a formal and systematic way. Within software development, the adoption of a structured methodology may go some way toward providing the basis for a QMS but the QMS should go further in ensuring that responsibility is clearly established for the prescribed procedures and processes. An essential part of any QMS is a feedback loop to ensure continual improvement. This concept was made famous by Deming as the 'plan-do-check-act' wheel.

2.3.2.2 The Total Quality Management Philosophy

Total quality management (TQM) is a modern management philosophy which focuses on the concept of quality which it identifies as arising from the process central to the particular industry or project (Henderson-Sellers, 1991). It is described by Oakland as
"A method for ridding people's lives of wasted effort by involving everybody in the process for improving the effectiveness of work, so that results are achieved in less time."

TQM originated in the seminars presented to the Japanese by American experts after the Second World War. The subsequent success of Japanese industry has caused other nations to consider attempting to adopt some of the philosophies of TQM. The major emphasis of TQM is the interdependence of quality improvement and participative management or *"quality through participation"* (Trammell et al, 1992). The basic lesson of quality through participation is that people are most committed to standards of quality that they set for themselves.

Japan's Project SIGMA for the industrialisation of software in Japan has as one of its goals a fivefold increase in Japanese software production capability. This is particularly striking in the light of an analysis of Japanese software factories in which it was estimated that production capability was already six times greater than equivalent efforts in the United States (Henderson-Sellers, 1991).

While the philosophy of TQM has developed essentially for the industrial and commercial environments, Henderson-Sellers (1991) explores parallels between TQM and object-oriented software development. This work is motivated by his belief that:

"the application of object-oriented ideas to the management of software development is also aimed at improving the quality of the finished product."

Both the TQM and the OO philosophies require a different mindset for their essentially holistic design. Both are evolutionary rather than revolutionary. Current software development practices essentially have a 'build fast, test later' attitude. Even with subroutines designed to be archived and reused in later projects there is no quality assurance that they will perform adequately in the new environment. In developing object-oriented library classes it is vital that quality be built into each class module before it is accepted into the reuse library.

Deming's 14 points of TQM (Deming, 1986) are :

1. Create constancy of purpose.
2. Adopt the new TQM philosophy.
3. Evaluate the system objectively and quantitatively.
4. Don't award business on price tag, rather on quality.
5. Aim for constant improvement.
6. Institute on-the-job training.
7. Institute leadership rather than control.
8. Drive out fear of punishment.
9. Break down inter-departmental barriers and rivalries.
10. Eliminate slogans (which are not usually achievable).
11. Eliminate numerical goals and objectives.
12. Give workers pride in their work.
13. Institute programmes of self-improvement.
14. Involve everyone.

Some of these 14 points can be examined directly in terms of parallels in the object-oriented philosophy. With the increasing emergence of the OO philosophy into the corporate consciousness, it is important that a leadership decision is made to adopt the new ideas (point 2) and embody them within a strategic planning framework (point 1). Continuous monitoring of OO classes during the development process permits continuous assessment to be made (point 3). Classes destined for generic libraries can be both closed (in the sense of being reliable classes available to the general software developer) and open (in the sense of being easily improved) (point 5). The ability for such built-in quality must mean that the modules of code developed in such a quality managed environment will be purchased on an evaluation based on this stated quality, rather than on price (point 4). It might also be anticipated that software developers of truly generic and useful classes will indeed take pride in a job well done (point 12). The arguments on the best method of introduction of both TQM and OO also have strong similarities. Although top commitment is necessary (point 7), it is also important that practitioners understand and sympathize with these new approaches (point 14). Consequently, education and training policies are necessary throughout the company (points 6 & 13). Points 8-11 are less easily paralleled in the object-oriented approach. They are more easily addressed within the context of the management philosophy utilized. It is, however, debatable whether a total commitment to these new philosophies should be made immediately, or whether it is more realistic to replace the management philosophy incrementally. It certainly does appear, however, that both philosophies can benefit from further mutual interaction.

2.3.2.3 The Quality Improvement Program (QIP)

This refers to programmes designed to improved quality based on the introduction or refinement of a QMS. The emphasis is on improvement rather than on monitoring the current state of affairs.

2.3.3 Quality within the Software Development Process

A software systems development life cycle model structures the development process into a number of phases. There are a number of different models currently favoured by various developers. Probably the most well-known is the Waterfall Model. Other models (Gillies, 1992) are the Code-and-Fix Model, the Iterative or Evolutionary Model and the Transform Model. A model which is also important, is the Rapid Prototyping Model. Each of these lifecycle models is reviewed below.

The Waterfall Model generally follows the sequence of

a. Requirements Analysis
b. System Specification
c. High Level Design
d. Detailed Design
e. Implementation
f. Maintenance

Although this approach has become the basis for most software acquisition standards it is by no means infallible. A major problem with the Waterfall Model is the emphasis on fully elaborated baseline documents as completion criteria for the requirements, specification and design phases. Although this is sometimes the best way to proceed it is not very effective for many classes of software and leads to elaborate specifications of poorly understood user interfaces and decision support functions resulting in the development of unusable code.

The Code-and-Fix model consists of two stages :

a. Write some code
b. Fix any problems in the code

Except perhaps in cases where the program is very small and easy to understand, this model produces software of poor quality. After a number of fixes, the resulting code is poorly structured, the program often does not really conform to the users' needs as no requirements analysis is done, and the system is difficult to maintain and modify.

In the Iterative or Evolutionary Model, the development stages consist of expanding increments of an operational software product, with each new iteration being determined by operational experience. The major problem with this approach is that it is based on the assumption that the user's operational system will be flexible enough to accommodate unplanned evolution paths and the result is often difficult to distinguish from code produced using the Code-and-Fix method.

The Transform Model assumes the existence of a capability to automatically convert a formal specification of a software product into a program which satisfies the specification. This model consists of the following stages:

a. Formal system specification
b. Automatic transformation of the specification into code.
c. Optimise the code.
d. Test the resulting product.
e. Adjust the specification based on the results of the previous stage and repeat from stage b.

The Transform Model is only suitable for small products in a few limited areas, however, and it also makes the, possibly incorrect, assumption that the users' operational system will be flexible enough to support unplanned evolution paths.

In the Rapid Prototyping Model, a working model that is functionally equivalent to a subset of the perceived finished product, called a prototype, is built. The client is then allowed to interact with this model and make suggestions. The prototype is then revised and the client experiments once more. Once the client is satisfied that the prototype does most of what is required, the developers can draw up a specification document confident that they know the client's real needs. Prototypes have potential use for many facets of SQA. According to Staknis (1990) they can:

- provide a tangible basis for evaluating quality of concept early in the life cycle.
- provide empirical evidence of an algorithm's efficiency.
- serve to familiarize quality assurance personnel with system requirements.

Prototypes can be used as vehicles to develop and validate testing procedures, they provide insight into how to design a system for ease of testing. Other advantages claimed for prototyping are :

- Increased user involvement leads to greater user satisfaction.
- A reduction in the number of errors in the resulting system specification.
- An improvement in communication between the software developers and the users.
- Critical attributes are identified earlier.
- Problems within the development team are exposed sooner.
- Early delivery of a tangible piece of code can gain credibility.

One of the biggest problems with prototyping is knowing when to stop.

Boehm (1988) proposes a Spiral Model for software development which he believes incorporates many of the strengths of the other models while resolving many of their difficulties. The underlying concept behind this model is that each cycle in the development process involves a progression that addresses the same sequence of steps for each portion of the product and for each of its levels of elaboration.

The major advantages of the spiral model are :

a. Early focus on options involving the reuse of existing software.
   This is definitely a quality assurance advantage as code reuse, when properly controlled, definitely promotes productivity as well as system stability.
b. It promotes preparation for later modification to the system.
c. Because all types of objectives and constraints are identified during each round of the spiral, software quality control is facilitated.
d. It focuses on eliminating errors and unattractive alternatives early.
e. The risk-driven approach means that for each project activity and resource usage in a project a certain amount of time and effort is allocated depending on that specific project.
f. A viable framework for integrated hardware and software system development is provided.

According to Boehm there are only three major difficulties in applying the spiral model. These are :

a. Matching to contract software. The spiral model works well on internal software, the development of which has a great deal of flexibility and freedom to accommodate stage-by-stage commitments. Progress has been made in establishing more flexible contract mechanisms but these still need to be worked on.
b. Relying on risk assessment expertise. The risk driven approach relies for its success on the expertise of the people using it. Inexperienced developers may produce insufficient specifications, while specifications designed by an experienced team for a non-experienced team of implementers will need additional documentation to keep the less-experienced team from going astray.
c. The need for further elaboration of spiral model steps. For example, there is a need for more detailed definitions of the nature of spiral model specifications and milestones.

However, the risk-driven nature of the spiral model appears to be more adaptable than the other approaches, especially for large, complex software systems.
Du Plessis and van der Walt (1992) have revised the Spiral Model to complement object-oriented development and it is this revised model which will be used here to delineate quality achievement within the software development process. This model proposes a three-dimensional view of the software development process. Firstly, software development is viewed at three levels of abstraction:

a. **Universal Level** A global, management view of the project based on the cycles of the spiral.
b. **Worldly Level** At this level the planning and scheduling of tasks, money and resources is performed.
c. **Atomic Level** Actual data and object design and software development takes place at this level.

Quality assurance plays a role at each of the levels. At the Universal Level, the Project Manager must produce a product which is reliable and maintainable and which satisfies the user. He must make sure that the project time and budget constraints are correctly estimated.
At the Worldly Level, appropriate balances between the various software criteria need to be ascertained, for example, should the system be very fast, possibly at the expense of some error handling, or it is more important that the system be very stable? Development standards also need to be identified and considered. All of this must be achieved within the time, resource and budget constraints set at the universal level.
At the Atomic Level, system verification and validation is important. Criteria such as reusability, flexibility and modifiability must all be optimised when objects are designed.

The Model can also be viewed in terms of four quadrants (figure 2.4):
a. **Issue Formulation** during which the objectives, needs, constraints and alternatives are considered with respect to satisfying the stated objectives for a cycle. A strategy is then formulated enabling the objectives to be reached. Alternative strategies and any applicable constraints should also be identified. Work in the quadrant culminates with a strategy definition review in which the objectives and strategies are reviewed.

b. **Risk Analysis and Evaluation of Alternatives** This quadrant can be divided into 3 sub-phases:

1) Risk Analysis — the proposed strategies are evaluated and the risks involved are identified.
2) Risk aversion planning — alternatives are evaluated in terms of the objectives and constraints in order to reduce the risks involved.
3) Prototyping — various modelling techniques, including prototyping, may be used to test areas of risk.

This quadrant culminates in a commitment to a specific strategy.

c. **Development** The development activities and tasks for a cycle are performed. Provision is made for evolutionary development as this quadrant can be visited for each module of code or sub-system under development.

d. **Review/Planning** An assessment is made of progress for the cycle and a decision is taken as to whether to continue the process or not. This quadrant may be divided into two main sub-phases:

1) Evaluation - the deliverables produced in the development quadrant are verified against the specifications and a decision is made whether to update the project baseline. Quality related activities such as code walkthroughs are performed. The evaluation phase ends when the product development review is done. The result of this review is the acceptance of the product.
2) Planning for the next cycle of development is done. A review of the work breakdown structure and cost breakdown structure is done. Software cost estimations are updated and the schedule for the next phase is decided upon. The planning phase ends when a software development plan for the next cycle is presented to the customer and the commitment to continue with the next cycle is obtained.

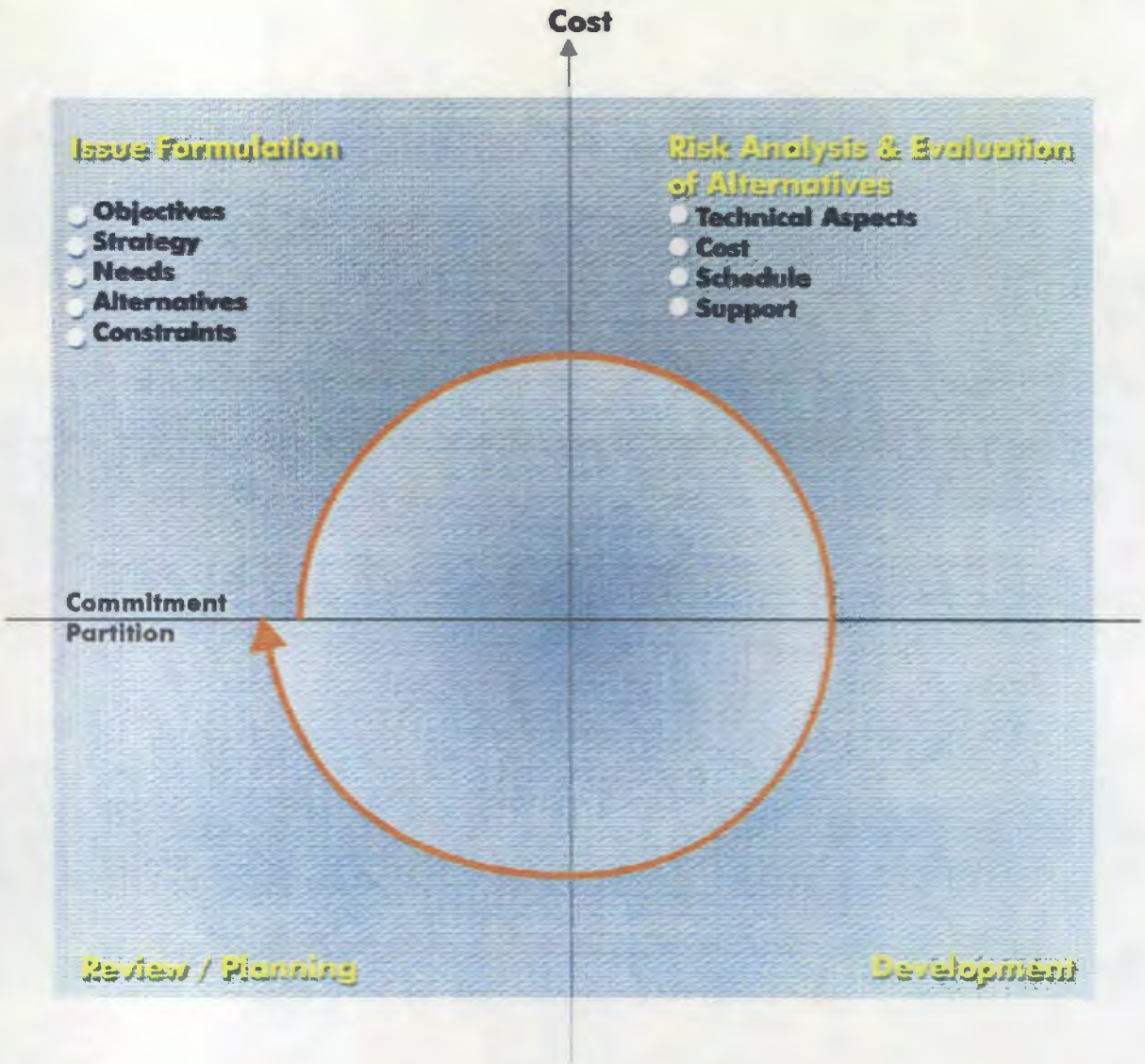

Fig. 2.4 **Quadrants of the Revised Spiral Model**

Finally, the five cycles of the Spiral Model are summarised below (figure 2.5):

a. Feasibility - This cycle starts with the formulation of a problem statement and culminates with a Project Proposal. In the issue formulation quadrant a problem statement is formulated including a description of the implementation environment and any constraints which are relevant to the project. During problem analysis and evaluation, alternative system configurations are evaluated. A feasibility study is conducted in the development quadrant resulting in the first cost benefit analysis, technical feasibility evaluation and an investigation of any legal implications that might be involved.

The feasibility cycle is traversed once only. As soon as the Project Proposal is accepted and authorized by management the project enters the next cycle.

b. Architecture - In this cycle the top level system software and hardware architectures are determined. Only after this has been completed should a final cost estimate be done and a tender for the completion of the system be considered. The Architecture Cycle starts with the formulation of objectives for the cycle and concludes with the planning of the analysis phase as well as that for the rest of the system. The work breakdown structure, cost break down structure and schedules are updated and a.Project Proposal for the development of the rest of the system is compiled. The project management plan for the rest of the project is also defined. This cycle ends when the client accept the revised Project Proposal. If, during later cycles, it is realized that the specifications arrived at during this cycle will not suffice, this cycle may be revisited.

c. Analysis - The main guidelines for the analysis strategy are defined in the Architectural cycle so the top level class diagrams should have been created before embarking on this cycle. This leaves the analysis cycle with the task of completing the object-oriented analysis using the diagrams defined during the previous cycle. Risk analysis is done for defined strategies and prototyping of the top level class structure may be done to ensure the feasibility of the strategy. Identified risk areas may also be prototyped to resolve uncertainties. An object model, dynamic model and functional model of the system should be defined in the development quadrant of this cycle. Finally, in the fourth quadrant, the three models are reviewed to ensure that they are consistent and complete.

d. Design - The system is designed and sub-systems are developed during this cycle. In the first quadrant the objectives for the design cycle are reviewed and a design strategy formulated. The classes which form the kernel of the system are developed and testable modules are produced. This is followed by risk analysis and risk aversion plans are made. Prototyping of high risk aspects is done to enhance understanding of the system. Because object-oriented development is an incremental process, the steps in the development quadrant may be reiterated before a sub-system is completed. Once a sub-system is complete it is verified. A product development review is performed during which the code is either accepted or resubmitted to a previous cycle.

e. **Implementation** - In the Issue Formulation quadrant, various strategies for system integration and integration testing are defined. Ways in which the hardware of the system could be installed are identified and a training program for users is created. In the next quadrant, the risks involved in the different integration strategies are identified and integration tests are evaluated to ensure their adequacy. The hardware layout and software packaging should be reviewed and the training program is assessed to ensure that it covers all aspects of the system. In the development quadrant system integration is performed and tested. This may reveal incompatibilities between interfaces requiring the design cycle to be repeated. Progress is continually monitored. Finally, the system acceptance testing is performed and if the customer is satisfied that the system executes according to the specifications, the system is considered to be complete.

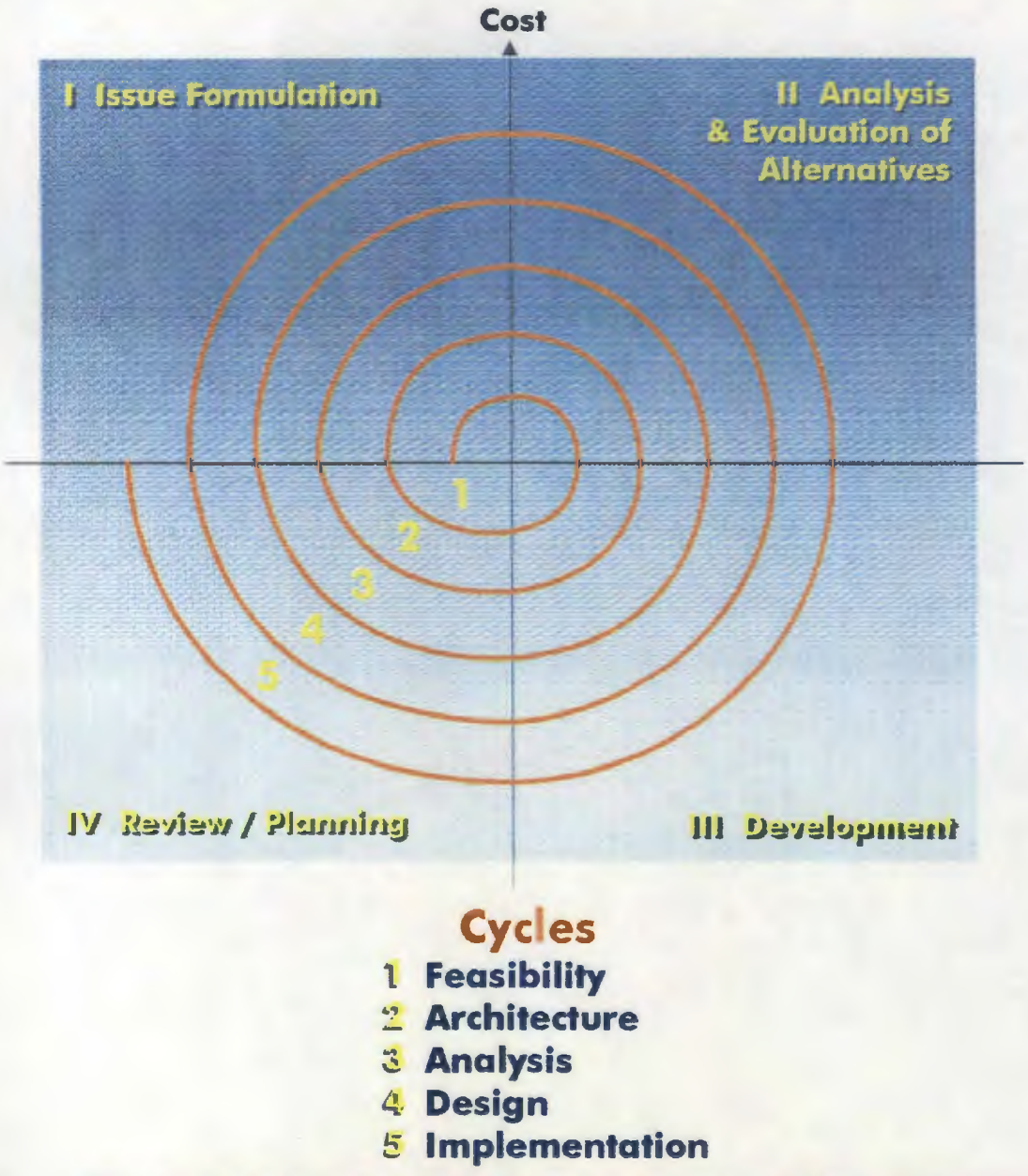

Fig. 2.5 **Revised Spiral Model for Object-Oriented Development**

2.3.4 Metrics for Measuring the Quality of Project Management

The quintessential issue of project management is identifying the activities needed to complete the project and then assuring that resources will be available when they are needed. The dominant cost estimation fault modality, according to Dunn (1990), is underestimation. He stresses that managers need to do more than just estimate the total number of labour-hours that will be used. The main pitfall is that the manager rarely knows what the project really consists of at its inception. The software requirements are only faintly defined and one has only a vague notion of the structure that will emerge.

The development and validation of software metrics is expected to provide many benefits in the form of producing accurate cost and scheduling estimations providing valuable aid to managers in making decisions with regard to budget and personnel allocations and in making reliable bids for contract competition. Metrics are needed for the following areas:

a. Project cost estimation.
b. Measuring the productivity of new tools and techniques.
c. Establishing productivity trends.
d. Improving software quality.
e. Forecasting staffing needs.
f. Anticipating maintenance requirements.
g. Estimating software production time.

Metrics are needed in the early stages of the software life cycle in order that they are able to impact on the development process.

Software size estimation is an important metric in that it relates directly to the estimated cost, time and staffing requirements for the development of a software system. Luiz Laranjeira (1990) mentions two techniques that have been proposed for the purpose of predicting software size : subjective techniques and objective models.

Subjective techniques rely on expert judgement based on analogies with previous projects, experience and the intuition of the estimators. Experience has shown, however, that expert judgement varies widely and usually cannot provide estimates with the required level of accuracy. Also, subjective estimates tend towards underestimation for a number of reasons.

In a specification level objective model the size of a system is generally expressed as a function of some known qualities that reflect characteristics of the system such as the number of input and output files, the number of internal logical files or the number of enquiries. The quantities upon which these estimates are based are usually related to the interactions between the system and the environment. It is obvious, however, that programs having the same type and number of interactions with the environment might have totally different sizes. Complexity of a system also does not necessarily relate to its size. Finally, most of these models were developed for use in banking and business applications and are not really applicable for other types of systems.

An implementation level objective model attempts to express system size as a function of characteristics whose values can be determined at the detailed design stage, before actual coding begins. This is usually achieved by measuring the number of operations, unique variables and constants in the program. Although it would be expected that more accurate estimates would be obtained from the use of these models due to the extra knowledge of the project which is available after the detailed design phase, this has not tended to be the case. The main reason for this is that, as before, programs with the same number of variables may differ considerably in size and complexity.

It would appear, therefore, that more specific knowledge about the nature of the software system is necessary in order to estimate its size. Functional requirements modelling has become a vital phase in the software development process. According to Laranjeira (1990), the main role of a functional model is to provide the following:

Abstraction	representing several objects while suppressing details.
Partition	dividing the system into components, allowing one to concentrate on each component separately.
Projection	representing the entire system with respect to only one set of its properties.

2.3.5 Techniques for Testing the Correctness of Software

2.3.5.1 The purpose of Testing

The purpose of testing software systems is to ascertain their reliability. If a system is tested and works as intended, then confidence in the system would tend to be high. However, this high confidence may not be justified. Graham (1992) states that:

"In order to assess software quality effectively, it is essential that the quality of the testing itself be taken into consideration."

The first thing software quality engineers want to do is to cooperate with the programmers in identifying discrete test stages, each directed to a specific objective or set of objectives. This provides a basis for visibility of progress, while at the same time lending assistance to the planning and scheduling tasks. Most importantly, a well-considered incremental test strategy permits a variety of types of testing diverse enough to provide confidence in the total efficacy of the testing process.
Testing activities are included in the quality control plan (QCP) which is used to drive the quality to be built into the software product. The QCP addresses issues such as the testing strategy, test tools needed, acceptance criteria, what reviews are to be performed, risk management, user involvement, what training is needed in testing techniques, and what test metrics will be used.

A typical set of testing stages used for large software systems includes:

1. Object or module testing, where each unit is tested individually, affording the maximum capability of exercising each of its segments and each aspect of its function.
2. Integration testing, where the units comprising each of the several sub-systems constituting the overall software system are welded together, providing an opportunity to find defects in the structure of each subsystem and in the interfaces of its units.
3. System testing is the stage in which the development people demonstrate that the product is sufficiently robust and defect-free to warrent the next testing stage.
4. Qualification testing represents a major departure from the preceding tests. Now the responsibility rests with the software quality engineers who must demonstrate that the system is ready for operational use. System testing has demonstrated that the software is sound; now it must be shown that it can be released.

Quality assurance ensures that tests are developed according to the prevailing standards and at the right times, and performs checks on test quality. It also checks that the testing being planned or performed is effective in achieving the test objectives as set out in the test strategy. Software quality assurance can be enhanced using various testing techniques, both manual and automatic.

2.3.5.2 Manual Techniques

Manual techniques include walkthroughs, inspections, test and evaluation, and formal verification and validation. These are expensive and time-consuming activities.

Walkthroughs

The participants in a walkthrough should be four to six people with representatives from the specifications team and a client representative as well as a representative of the team which will perform the next phase of the development. These representatives should preferably be experienced senior technical staff. The purpose of a walkthrough is to record faults found for later correction. Walkthroughs may be participant-driven or document-driven. The latter is likely to be more thorough and usually leads to the detection of more faults. Walkthroughs consist of two steps: preparation, followed by a group analysis of the document.

Inspections

The primary purpose of an inspection is to remove defects as early as possible in the development process. Secondary purposes are :

- to provide traceability of requirements to design.
- to provide a technically correct base for the next phase of development.
- to increase programming quality.
- to increase product quality at delivery.
- to achieve a lower life-cycle cost.
- to increase the effectiveness of the test activity.
- to provide a first indication of program maintainability.
- to encourage entry/exit-criteria software management.

An inspection is a five-step process: overview, preparation, inspection, rework and follow-up. A checklist of potential faults is an essential component of an inspection while a record of fault statistics is also important. An inspection takes much longer than a walkthrough but they have been shown to be very powerful and cost-effective tools for fault detection.

Verification and Validation

Verification and Validation (V&V) is a collection of analysis and testing activities across the full life-cycle. Schulmeyer et al (1987) define V&V as follows:
"Verification and validation is the systematic process of analysing, evaluating and testing system and software documentation and code to ensure the highest possible quality, reliability and satisfaction of system needs and objectives."

It uncovers high-risk errors early giving the design team time to evolve comprehensive solutions, and evaluates the product against system requirements. The V&V group may perform in-depth evaluations like rederiving algorithms from basic principles, computing timing data to verify response-time requirements and developing control-flow diagrams to identify missing or erroneous requirements. V&V standards have been developed which are applicable to many types of software.

Test and Evaluation

Software testing and evaluation (T&E) can be defined as
"the process of exercising complete programs to judge their quality through the fulfillment of specified requirements" (Schulmeyer et al, 1987).
The difference between V&V and T&E is that V&V can be accomplished without testing the software. The essence of V&V lies in the traceability of requirements.

2.3.5.3 Automatic Techniques

When humans test software manually by entering data through a user interface, the need to run a large number of test cases can present a serious obstacle. Because humans are slow compared with computers they introduce a bottleneck into the testing process. To remove this bottleneck, testing should be as automated as possible. Although some of the methods described here can be performed on paper, they can be automated.

Correctness Proofs

A correctness proof is a mathematical technique for showing that a product satisfies its specifications. However, one can never be sure that the specifications are correct nor can one be certain that a correctness proof is correct. Sage and Palmer (1990) feel that:
"Formal proofs of software correctness generally rely on both structural and functional constraints and they will usually necessitate very unrealistic assumptions in order to render the mathematics associated with a formal proof tractable."
Large-scale developments are normally too complex for the use of these proofs as it is almost as difficult to prove the correctness of the method. Shach (1991) maintains that if the cost of proving software correct is less than the probable cost if the product fails, then the product should be proved. He qualifies this by mentioning that correctness proving on its own is not enough and should be viewed simply as a component of the set of techniques that must be utilized together to check that a product is correct.

Prototyping

The problem of checking results presents a conundrum for automated testing: it is not possible to check the software's results by comparison unless the results are already available, but software is needed to produce the results. One answer to this problem is to use a prototype to produce the results needed for comparison. There may be a problem if he results produced by the prototype and by the final system have different forms. Also, prototypes are not usually used to generate meaningful results, rather they are utilized as a means to define the content and appearance of the software. Therefore, this method is meaningful only in those cases where the functionality rather than the form of the system is the focus of the prototype. There are a number of methods that may be employed when using prototyping as a means of automated testing. The prototype can be used to pregenerate results and to store these results in a file of input-result pairs. The inputs can then be passed to the system being tested and the results compared with the corresponding pregenerated results. Alternatively, a testing program could call both the prototype and the system to be tested concurrently with the same input and dynamically compare the results.

2.3.5.4 Testing Object-Oriented Programs

According to Smith and Robson (1992), the difference between testing an object-oriented program and a standard sequential program is that OO programs are not executed sequentially and routines from a class can be combined in an arbitrary order. Thus, the testing process becomes a searching problem. The objects and classes of the object-oriented model contain routines and possible data structures that contain the state of the object. There is, therefore, no sequential input, process, output model into which testing methods can be adapted. Smith and Robson (1992) propose several strategies for testing object-oriented programs. These include:

- Minimal: this provides for a minimal algorithm to search for errors in the class under test that can be overridden by child classes which inherit from it.
- Exhaustive: this strategy exercises all legal combinations of routines supported by an object, to a sufficiently great depth of combinations.
- Tester Guided: this strategy includes human guidence in the form of either creating subsets of the routines to be exercised together, or suggesting particular combinations of operations to try in certain sequences.
- Inheritance: the intention here is to use regression analysis to determine which routines should be tested and then to perform the tests based on how the superclass was successfully tested.
- Memory: In object-oriented programs, many objects are created and deleted at runtime. One test strategy flowing from this is to test whether the programmer deallocates objects when they are no longer needed.

- Data flow: although OO programs have a different flow through them than traditional programs, the use of traditional data flow techniques can be adapted into a dynamic approach for OO programs.
- Identity: A strategy can be defined that searches for pairs (or more) of routines that leave the state as it was originally, before any routines were invoked. This can be reported to the tester who can determine if the routines listed should perform in such a way.

2.3.6 Techniques for producing quality software

2.3.6.1 Appointment of an SQA Team

In large projects it may be worthwhile to appoint a team primarily responsible for software quality assurance. The duties of such a team would include:

a. The development and implementation of quality assurance standards for the project.
b. The development and implementation of metrics, testing tools and other quality assurance techniques.
c. The implementation of the resulting plan.
d. Producing a final quality assurance report.

A software quality assurance plan (SQAP) would have the following essential components:

a. Identification of the scope and purpose of the plan.
b. Identification of the organizational structure for implementing the plan.
c. Identification of the documents that need to be prepared, and methods to determine the quality and adequacy of this documentation.
d. Identification of metrics, standards, procedures and practices, including reviews and audits, that will be used in implementing the plan.
e. Identification of methods that will be used in collecting, maintaining and recording quality assurance information.
f. Implementation of each of these.

Trammel & Poore (1992) propose designing a 'rule set' as an operational definition of software quality for a given working environment. In their paper, they give a sample software quality rule set based on a VAX FORTRAN environment:

a. Meet the stated objective.
b. Make modules robust.
c. Provide an overall description of the module in the header.
d. Provide section comments in the body.
e. Follow a consistent standard.
f. Use modular design.
g. Use structured programming.
h. Use logic that is easy to follow.
i. Design code to be flexible, for maximum usage with minimum modification.

A rule set should define the standard practices through which the work group will achieve the highest level of quality of which it is capable at a given point in time. As capability increases, the rule set can be strengthened to represent higher quality goals. To design a rule set a 'jury' of seasoned practitioners from an organization's software staff rank a randomly selected sample of modules from the organization's software inventory. These ranked modules are then statistically analysed to search for a metric that correlates highly with the jury's ranking and this metric is then included in a 'rule set' that reflects the group's operational sense of quality.

2.3.6.2 CASE Tools

During the development of a software product, a number of operations and functions are carried out. Unfortunately, none of these activities can yet be fully automated and performed by a computer with no human intervention but computers are able to provide assistance every step of the way. CASE stands for Computer-Aided Software Engineering.

The simplest form of CASE is the software tool, a product that assists in just one aspect of the production of software. CASE tools are computer based tools to assist in the software engineering process. The advantages of using CASE tools include :

1. Increased productivity.
2. Consistency across development teams and projects.
3. Methodology automation ensuring that the developer sticks to the underlying methodology.
4. Varying degrees of automated documentation can be provided by CASE tools.
5. Improved maintainability of a system.
6. Lower development costs.
7. Improved product performance.

In addition to the above, Schulmeyer et al (1987) feel that the need for a fully integrated CASE environment is also highlighted by two other factors:

1. The cost of software continues to rise. Software costs have, in fact, reached crisis proportions.
2. The quality of software has also suffered due to the fact that the rising complexity of most large software programs makes it more difficult to know what is actually going on during development. Development takes the form of concurrent processes which all come together for the first time during the software integration phase. Visibility needs to be exploited on the front-end of the software development process.

However, most CASE tools do not provide complete life-cycle support, usually failing to provide facilities to automatically produce code from the design. This causes the CASE tool to lose one of its greatest advantages, the integrity of the design from inception right down to the finished code. On the other hand, although it is easier to maintain a well-designed piece of software, if the code does happen to be automatically generated, it will have to be maintained within the CASE environment in which it was first produced as the automatically generated variable names would tend to make the source code unintelligible, and this can be very limiting. Also, code generated using CASE tools is unlikely to be efficient. Thus, although CASE tools do seem to enhance some aspects of the development of software, they are by no means the panacea they are sometimes purported to be.

One of the major problems for CASE is the lack of a standardisation scheme to integrate tools. No clear, all encompassing, workable set of requirements exists from which a universally compatible integration mechanism may be developed. Schulmeyer et al (1987) define integration for CASE as one or a combination of the following levels:

Level-1 : Presentation Integration is a mechanism for invoking CASE tools and tool functions through a common user interface.

Level-2 : Control Integration uses 'triggers' to control the sequential execution of tasks through the use of integrated CASE tools based on information reporting among the tools. This might occur when a requirements tool executes an analysis task and then triggers the design tool.

Level-3 : Data Integration is a mechanism which permits global sharing of data among integrated CASE tools and includes access to both local and public data storage areas. The current problem in data integration is that of data incompatibility when moving data between tools. The rules and relationships surrounding databases are controlled by the tools creating the data. This can result in data mismatches or lost meanings when data is transferred to secondary tools.

Level-4 : Full Integration is a combination of the above three levels and is generally referred to as an Integrated Project Support Environment (IPSE).

Types of CASE tools

Complexity Analyzer	is used to determine the complexity of software design or code.
Database analyzer	is used to investigate the structure of flow within a data base to determine whether performance goals can be realized.
Logic Analyser	is used to inspect the use of control logic within a program, determine if it is proper and mechanize the design.
Reliability Models	are automated packages used to assess the probability with which the software will perform its required functions during a stated period of time.
Simulators	are used to represent certain features or functions of the behaviour of a physical or abstract system.
Standards Analyzer	a tool used to determine whether prescribed development standards have been followed.
Test Drivers	a tool to invoke an item under test, providing test input and reporting on test results.
Cause & Effect Graphs	a diagrammatic tool used to show cause and effect relationships for analysis purposes.
Comparator	is used to compare two software programs, files or data sets to identify the commonalties and/or differences.
Consistency Analyzer	employed to identify inconsistencies in conventions used in requirements, designs or programs.
Data Flow Analyzer	a tool used to determine if a data flow diagram is complete, consistent, and adheres to the established set of rules that govern its construction.

Fishbone Diagrams	a diagrammatic tool used to illustrate multiple relationships simultaneously.
Interface Analyzer	is used to determine if a range of variables is correct as they cross interface boundaries.
Metrics Analyzer	is used to collect, analyze and report the results of metrics quantification and analysis activities.
Requirements Tracer	an automated tool used to trace how the requirements were realized in the design and code.
Test Analyzer	a tool used to determine test case coverage.
Test Generator	is used to generate test cases directly from some specification.

Verifiable Languages

No documentation techniques have the potential to abet defect prevention more than those that employ verifiable languages which lend themselves to formal analyses of correctness such as proofs in first-order mathematics. These are not used casually, however, as much acquired skill is required of both the writer and the reader. As a half-way measure, structured requirements and design languages have gained considerable popularity. Although these lack the mathematical qualities of verifiable languages, they are machine readable and of formality sufficient for computer-generated reports, including lists of internal inconsistencies.

POKA-YOKE Devices

Schulmeyer et al define a Poka-Yoke device as:

"any device integrated into a process, at the point where a defect originates, to prevent the defect from occurring, thus mistake-proofing the process."

CASE tools can be effective poka-yoke devices in software development processes.

The Future of CASE

The advent of CASE tools has brought much enthusiasm to the software development community. The promise of a fully integrated software development environment looks feasible. The next generation in CASE technology foresees the enhancement of the data repository to utilize object-oriented design features for storing and retrieving processing and behaviour rules as well as data and other objects.

2.3.6.3 4GL

4GLs are powerful tools for expressing a problem solution in a language very close to the problem. Because they are application focussed they can be extremely powerful problem-solving tools for the class of problems for which they were designed. They are also usually user-programmable. The chief benefit of using them is productivity. Some further advantages, according to Glass (1992) are:

- Solutions coded for 4GLs that run in a number of computer settings will be portable.
- Succinct 4GLs result in code that is highly modifiable and understandable.
- Assuming the 4GL language processor itself is reliable, then 4GL solutions should be extremely reliable, especially because the programmer has far fewer opportunities to make errors than in a lower-level language.

A major trade-off, however, is in efficiency. A 4GL solution may run up to 100 times slower than its 3GL equivalent.

2.3.7 Software Defects and their Prevention

2.3.7.1 Zero Defect Software

Is zero defect software achievable? Errors are human and will always be made. According to Schulmeyer et al (1987), the secret to successful zero defect software is to isolate the errors humans make along the way and to remove them. Error detection can take place using any of the methods examined above. However, it is only through the people that the achievement of the zero defect software goal may be obtained.

Mistakes are caused by lack of knowledge and lack of attention. Knowledge can be measured and deficiencies corrected while lack of attention is an attitude problem and must be corrected by the person in question. Here follows a list on how defects in software come into being:

Defects created during the definition of requirements

Although any defect can lead to failure, defects traceable to the requirements phase of development are the most vexing since they often propagate to many pages of code. These defects usually fall into one of the following classes:

1. Incorrect reflection of operational environment in the allocation of elements of the solution to individual hardware and software constituents, something can get lost in translation.
2. Incomplete requirements.
3. Infeasible requirements.
4. Conflicting requirements.
5. Software requirements specification is inconsistent with other specifications - for example, the hardware environment in which the software is to operate may be overstated.
6. Improper description of the initial state of the system - all aspects of systems are not zero-valued at start-up, but this is what programmers will assume unless told otherwise.
7. Incorrect allocation of error - in translating system specifications to software specifications, it is not unusual to simply equate allowable software errors with allowable system errors.

Defects created during the design phase

1. Inadvertent range limitations.
2. Infinite loops.
3. Unauthorised or incorrect use of system resources - for example, the indiscriminate misuse of architectural features of the hardware processor, such as reserved registers.
4. Computational error improperly analysed - for example, designers often fail to account properly for the effect of truncation and rounding errors.
5. Conflicting data representations - for example, if one module processes in inches and another in centimetres.
6. Software interface anomalies - for example, parameters may be passed to modules that are not designed to catch them.
7. Defenseless design - the lack of adequate error traps or inadequate recovery action.
8. Inadequate exception handling.
9. Non-conformance to specified requirements.

Defects created during the coding phase

These are the classic programming bugs. For example:

- Misuse of variables.
- Mismatched parameter lists.
- Improper nesting of loops and branches.
- Undefined variables and initialisation defects.
- Infinite loops.
- Missing code.
- Unreachable code.
- Inverted predicates.
- Incomplete predicates.
- Failure to save or restore registers.
- Missing validity tests.
- Incorrectness of array components

In addition to these, we must include any failure to implement the design as documented.

Defects in Documentation and Installation

Incorrect or unclear information in user manuals can lead to operator errors and resulting failure. During installation, operating system parameters may not have set correctly.

Defects may be prevented or they may be removed. Obviously "prevention is better than cure", but cure is certainly better than nothing.

2.3.7.2 Defect Prevention

Defect prevention really means only two things: good programming and management support for good programmers. There is no more definitive way to foster good programming practices than by defining them as the standard way to do things. These then become programming standards. Standards that help prevent defects address the employment of tools, the forms of documentation, the handover of interim products from one development group to another, configuration management and the methods used for problem definition and program design. Standards that define methodology are directed to an attribute of computer programs closely linked with defect incidence: complexity.

The organised attack on complexity begins with the process of decomposition. Common to all thoughtful decomposition methods is the goal of a sensible, structured array of relatively small, highly cohesive and loosely coupled, modules. Decomposition applies to both the requirements definition and the design activities required for software development.

The frequently encountered difficulty of defining just what one wants software to do has been an historically infamous source of inadequate requirements specifications. To circumvent this, software departments are increasingly turning to the technique of rapid prototyping, which was discussed in more detail in section 2.2.2.

An interesting phenomenon, discovered by Dr Harlan Mills and mentioned in Keene (1991) was that the intolerance by the programmer of errors, and the expectation of making no errors is a great differentiator in producing the highest quality code. Apparently Dr Mills gave his students projects which they had to design and implement in code. The net score of each student was then adjusted by the number of trial runs the student had made. Apparently, after the first two weeks, all of the students' programs ran successfully on the first attempt. This indicates that perhaps one way around the software crisis is to be less tolerant of error in code.

2.3.7.3 Defect Removal

Not even the most optimistic software engineer believes that it is possible to construct a perfect defect-prevention program. The complexity and size of the current generation of programs continues to outpace the improvements in developmental method. According to Dunn (1990), some 35% to 50% of programming labour is spent on removing defects from software.

When we speak of defect removal, we really mean defect detection followed by a repair operation. By reviewing the requirements and design documents as well as the code, defect removal can take place long before the code is actually tested. One way in which reviews can take place is by means of inspections. There are three major types of inspection methods:

1. Judgement inspections that discover defects. Defects are generated by work and all judgement inspections do is to discover these defects.
2. Informative inspections that reduce defects. Information of a defect is fed back to the specific work process, which then corrects the process. This approach should gradually reduce production defect rates.
3. Source inspections that eliminate defects. They are methods that are based on discovering errors in conditions that give rise to defects and performing feedback and action at the error stage so as to keep these errors from turning into defects. Source inspections can be combined with poka-yoke measures to eliminate defects.

2.4 Summary and Conclusions

In order to achieve high quality software we must address two aspects of software design: the Technical Aspect and the Managerial Aspect. Each of these aspects concentrates on different but related viewpoints of software quality assurance. The Technical Aspect is concerned with how quality is defined and measured, while the Managerial Aspect is concerned with the attainment of quality in software. Both these aspects were covered in some detail in this chapter. Quality criteria, quality metrics, quality standards and quality techniques were all discussed under the Technical Aspect, while the Managerial Aspect concentrated on quality management systems, quality within the software development process, the quality of project management and techniques for testing the quality of software.

One theme that recurs time and time again is the fact that quality, like beauty, is very much in the eye of the beholder. What is important to one user may not be as important to another and therefore it is very difficult to prescribe a set of quality criteria or standards which are relevant to all users. Instead, the needs of each user must be assessed individually, within a basic methodology, and a quality assurance program designed specifically to meet the needs of that user. Each program will, however, consist of the same basic entities. Users will define the criteria they feel are relevant to their needs and assign them priorities, metrics will then be identified to measure the degree to which the criteria are satisfied. Standards will be set and basic quality assurance techniques followed. The managerial techniques employed will depend very much on the size of the project, the user's preferences and the developer's own organisation. There is no prescribed manner in which an organisation may decide which methodologies are best suited to it, rather the ones with which the management and their subordinates are most comfortable should be selected. Which tools will be used is very often subject to budgetary constraints, but again, there is no formula to help management decide which tools are most needed by their organisation. Rather, it is a decision based on studies of where the development process lacks in either productivity or quality.

Thus, it appears that there is no easy formula which can be applied to ensure a high quality finished product. Rather, a quality assurance methodology can simply provide the guidelines and ideas which, if adopted, may enable companies to improve the quality of their software.

CHAPTER 3

THE OBJECT-ORIENTED DESIGN METHODOLOGY

CHAPTER 3

THE OBJECT-ORIENTED DESIGN METHODOLOGY

3.1 Introduction

Because the project of which this study forms a part has adopted the object-oriented paradigm, and because it is proposed that the object-oriented methodology can ameliorate many problems associated with quality control in software systems development, object-orientation has been adopted for the purpose of this research. Therefore, the object- oriented design methodology will be described in more detail in this chapter.

According to Coad and Yourdon (1990) Object-oriented programming (OOP) was first used by the development group of the SIMULA language in the late 1960's. However, OO design and analysis as we know it today were not yet established and the procedural approach was still used. The procedural approach used functional decomposition to specify the tasks to be completed in order to solve a problem while the object-based approach concentrates on data specifications. The OO approach uses the relationships between objects as a fundamental part of the system architecture.

3.2 Object-Orientation - the Paradigm

Object oriented technology has evolved into a mature discipline for the design and implementation of software applications. Despite its popularity, however, there appears to be a certain amount of confusion regarding the term object-oriented. Because of its broad scope, the term is often misused, especially in marketing claims. Parsaye et al believe that the three most fundamental aspects of object-orientation are: abstract data typing (encapsulation), inheritance, and object identity (Parsaye et al, 1990), while Rumbaugh et al (1991) identify four aspects which they believe characterise an object-oriented approach: identity, classification, polymorphism and inheritance. Booch (1991) identifies the following major principles as essential for object-orientation: data abstraction, encapsulation, modularity, inheritance, classification and polymorphism. These principles will be discussed in more detail below.

3.2.1 Identity and Classification

The fundamental concept in object-oriented modelling and design is the object, which combines both data structure and behaviour in a single entity (Rumbaugh et al, 1991). Objects can be concrete such as a file in a filing system, or conceptual, such as a scheduling policy in a multiprocessing operating system. Each object has its own inherent identity. Objects with the same data structure and operations are grouped into a class. A class is an abstraction that describes properties important to an application. Any choice of classes is arbitrary and depends upon the application.

Each class describes a possibly infinite set of individual objects while each object is said to be an instance of its class (figure 3.1).

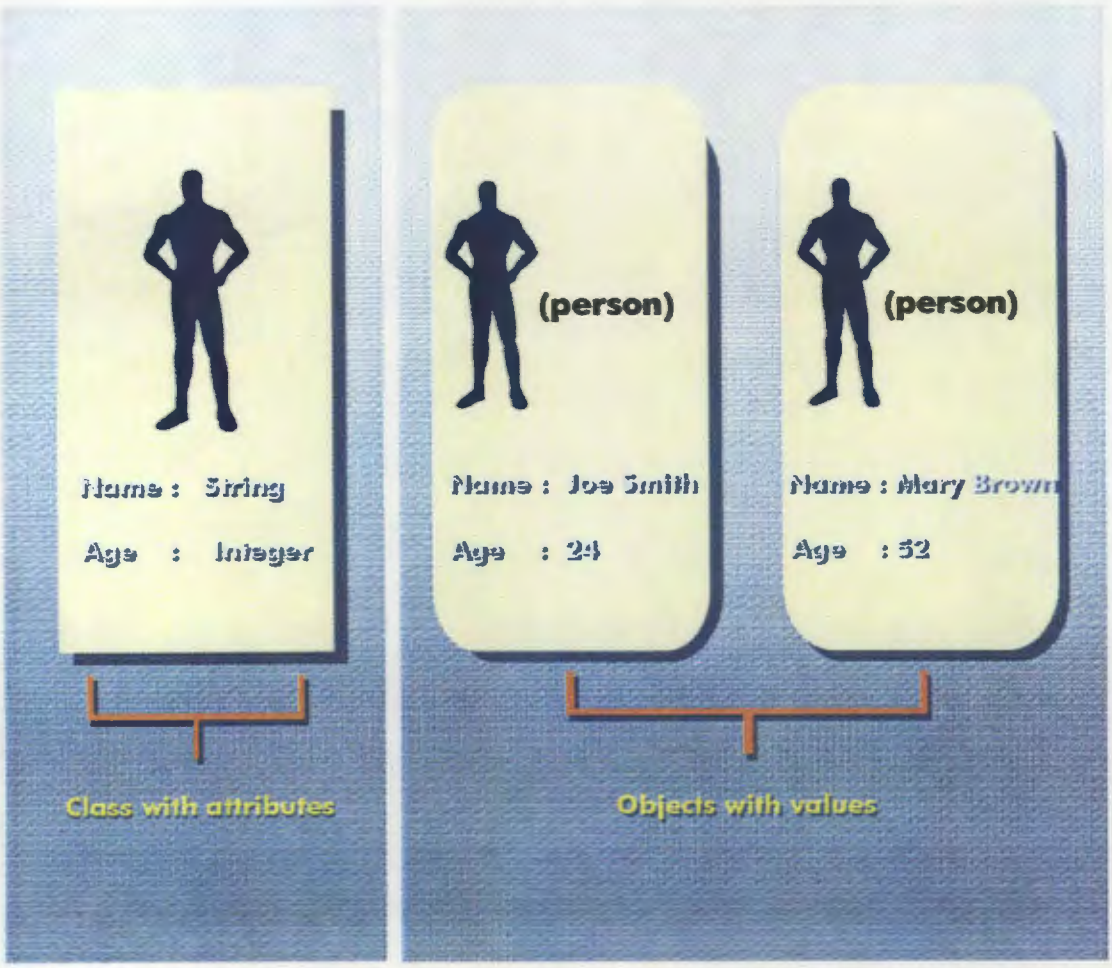

Fig. 3.1 **Classes and Objects**

3.2.2 Data Abstraction

Booch (1991) defines data abstraction as

"a denotation of the essential characteristics of an object which distinguish it from all other kinds of objects and provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

Rumbaugh et al (1991) state that the purpose of abstraction is to isolate those aspects which are important for some purpose and suppress those aspects which are unimportant.

Abstraction is used when the behaviour of an object is defined but the actual implementation of its behaviour is suppressed.

3.2.3 Encapsulation and Information Hiding

Various authors refer to encapsulation and information hiding as if they are synonymous (Rumbaugh et al, 1991; Booch, 1991 and Meyer, 1988). Encapsulation occurs when the user of an object is able to see the services provided by the object, but is unaware of how these services are actually implemented. In other words, the implementation details of an object are hidden from other objects. This means that the implementation of the object may be changed without affecting the applications using it.

3.2.4 Modularity

Booch (1991) defines modularity as a property of a system which has been decomposed into a set of cohesive, loosely coupled modules. In OOP this means the optimal physical packaging of classes and objects based on the design's logical structure.

3.2.5 Inheritance

Inheritance is a key reusability technique which is unique to the OO model. When a sub-class inherits from its super-class, it means that all operations defined in the super-class are also defined in the sub-class. Additional operations may be defined on the sub-class, and existing inherited operations may be redefined (figure 3.2).

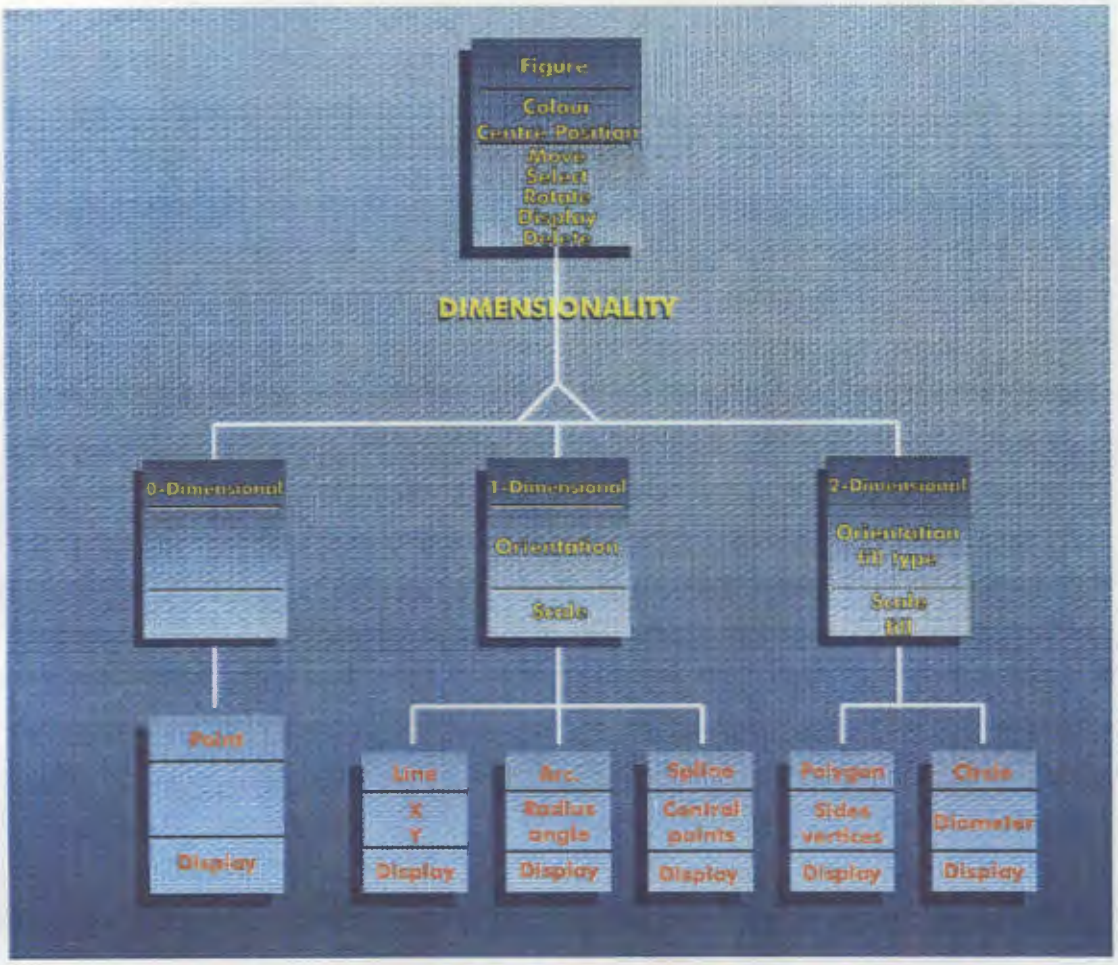

Fig. 3.2 **Inheritance in Object-Oriented Design**

3.2.6 Hierarchical Classification

A hierarchy may be defined as a ranking or ordering of abstractions. This hierarchy is obtained by the use of inheritance. Inheritance allows a sub-class to share the structure and/or behaviour defined in one or more super-classes.

3.2.7 Polymorphism

Meyer (1988) views polymorphism in OOP as the ability to refer at run-time to instances of various classes. Rumbaugh et al (1991) define a method as the implementation of an operation on a class. Various classes may have a method with the same name, but with different implementations. For example, the 'Draw' method may differ depending on whether the object is a circle, rectangle or polygon.

3.3 Object Modelling

3.3.1 Rumbaugh's Object Modelling Technique

The Object Modelling Technique proposed by Rumbach et al (1991) is adopted for the purpose of this study as it is the technique in use within the OOISEE project. Using the object modelling technique, software systems are described in terms of three different models, all of which must be tested in terms of quality.
These models are :

a. Object Model describes the objects in the system and their relationships.
b. Dynamic Model describes the interactions among objects in the system.
c. Functional Model describes the data transformations of the system.

This modelling technique inherently produces better quality systems because they are based on the underlying framework of the domain itself, rather than on ad-hoc functional requirements.

3.3.2 The Object Model

The two basic building blocks of the object model are objects and classes. An object may be described as a data entity with an associated set of operations which may execute on it, while a class may be viewed as a generic description of a group of similar objects. Although objects are loosely coupled, the classes used to implement the objects may be highly interdependent due to inheritance. A change to a parent class may have a ripple effect on all its sub-classes.
Rumbaugh et al (1991) propose that object diagrams, consisting of graphs where the nodes depict the object classes and the arcs depict the relationships between those classes, be used as a design tool. When large systems are developed, the set of object diagrams becomes large and the complexity of the system increases. It is, therefore, recommended that the system be divided into sub-systems before detailed object modelling is performed.

3.3.3 The Dynamic Model

This is concerned with the description of the sequence of events which occur in response to external stimuli. The flow of control, interactions and sequencing of operations in a system of concurrently active objects are described. Event traces and finite state diagrams are the most common techniques used for dynamic modelling.

3.3.4 The Functional Model

The Functional Model is concerned with how the output values in a computation are derived from the input values, without regard to the order in which these values are computed. Data flow diagrams are used to do this.

3.4 Object Modelling within a Spiral Life Cycle Model

A Revised Spiral Life Cycle Model for Object-Oriented development has been adopted as the software development process model for the purpose of this dissertation. This model consists of five cycles: the feasibility cycle, architecture cycle, analysis cycle, design cycle and implementation cycle. Strong emphasis is laid on risk analysis before any development work is done as this approach reduces the chance of project failure. Various reviews are held throughout the development life cycle as checkpoints to ensure control of the quality assurance process. If necessary, the QA process may regress to a previous cycle. A preliminary QA plan is developed during the feasibility cycle. During the architecture cycle, the foundation is laid for the object model, dynamic model and functional model and the hardware specification for the system is defined. The object, dynamic and functional models are refined during the analysis cycle, and detailed object, dynamic and functional models are developed during the design cycle.

3.5 Summary

In this chapter the most important aspects of the Object-Orientation paradigm have been identified. A brief description was given of the fundamental aspects of object-orientation and the object modelling technique defined by Rumbaugh et al (1991) was described.

CHAPTER 4

QUALITY ASSURANCE REFERENCE MODELS

CHAPTER 4

QUALITY ASSURANCE REFERENCE MODELS

4.1 Introduction

In order to compare quality in different situations, both qualitatively and quantitatively, it is necessary to establish a model of quality. There have been many models suggested for quality, most of which are hierarchical. In this chapter various different types of existing model are described and the state of quality assurance in general is discussed.

4.2 Existing Quality Assurance Reference Models

4.2.1 Hierarchical Models

The GE model proposed by McCall in 1977 (figure 4.1) is a hierarchical model of software quality and is intended for use by software developers during the software development process. The criteria were, however, chosen to reflect users' views as well as those of the software developer.

The model identifies three areas of work in software development :

- Product operation: the software must be easy to learn and operate, must work efficiently and produce the results that the user requires.
- Product revision: the product must be testable, maintainable and flexible. This is a very important area because it is generally considered to be the most costly part of software development.
- Product transition: software should be portable, reusable, and able to interface with other software.

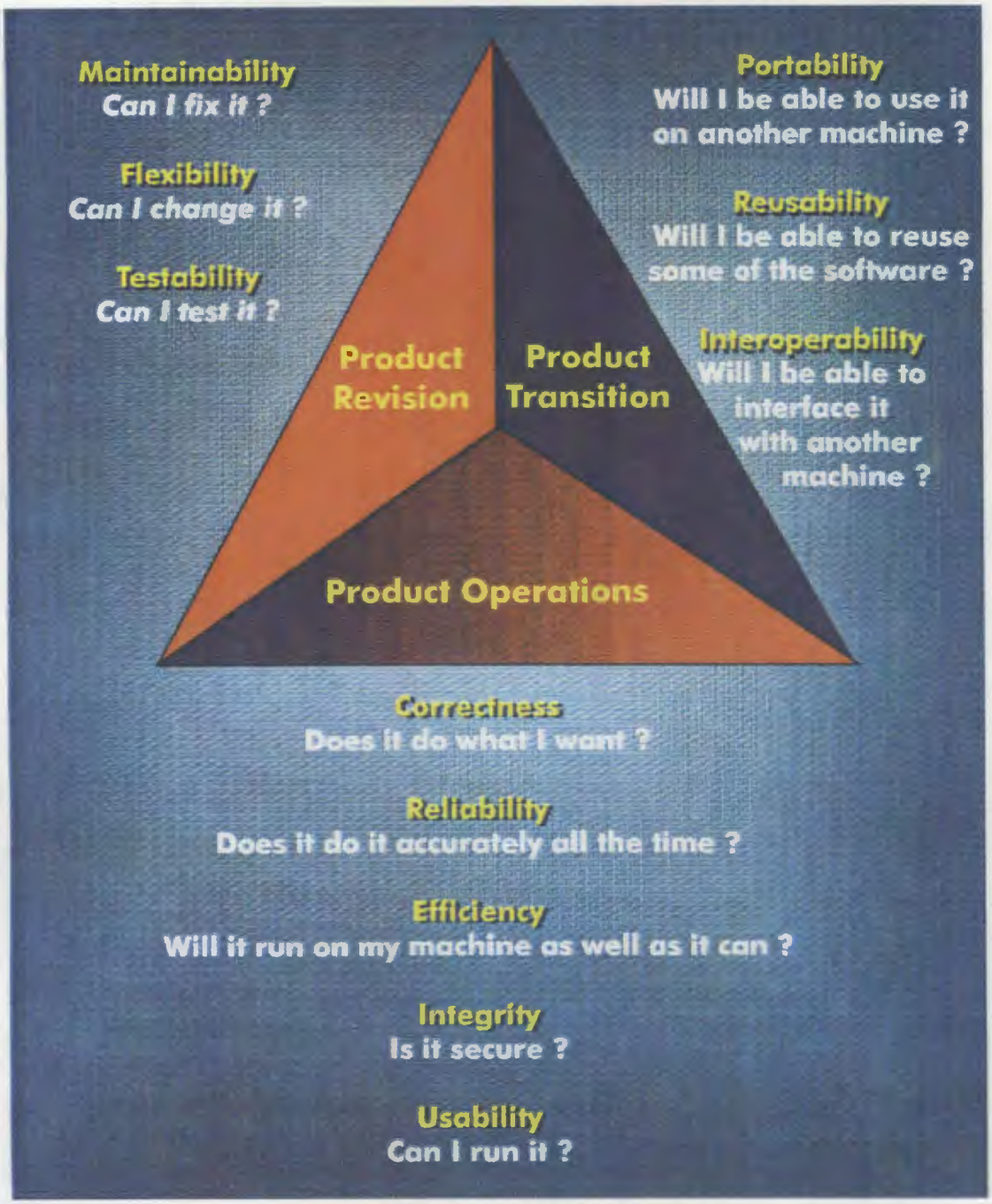

Fig. 4.1 **McCall's Model of Software Quality**

Boehm's model, defined in 1978, (figure 4.2) was to provide a set of "well-defined, well-differentiated characteristics of software quality" and is based on a much wider set of criteria than McCall's model.

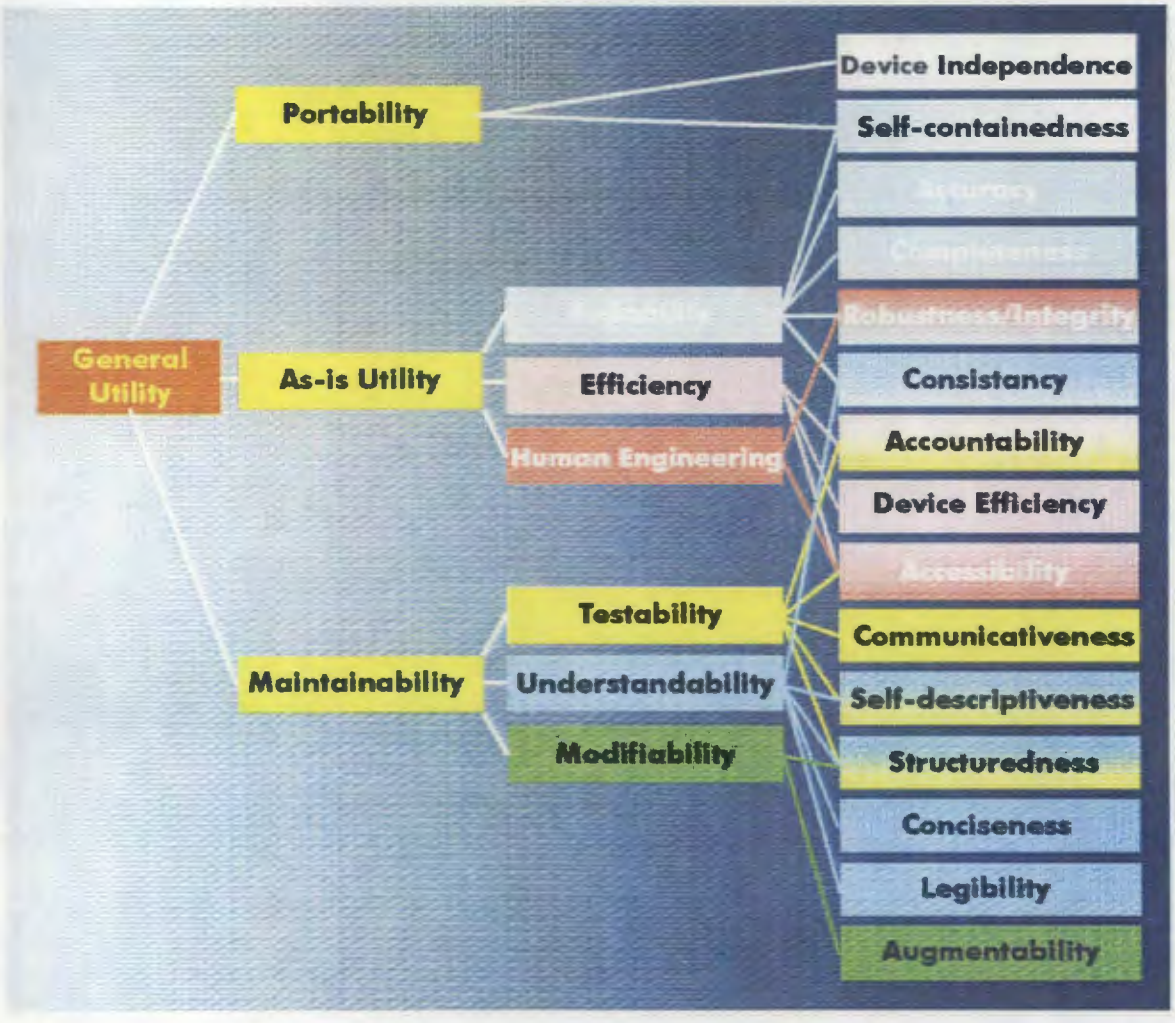

Fig. 4.2 **Boehm's Model of Quality**

Both models are hierarchical in nature and, although the quality criteria specified are supposedly based on a user's point of view, in fact, they are aimed more towards the software designer. Some quality criteria are defined in the negative, for example, Gillies (1992) defines maintainability as the absence of effort required to fix a bug in the software.

Other problems associated with these models are :

a. The distribution of metrics is not uniform amongst the quality criteria cited.
b. Hierarchical models cannot be tested or validated. It cannot be shown that the metrics accurately reflect the criteria.
c. The measurement of overall quality is achieved by a weighted summation of the characteristics. The resulting single 'figure of merit' is of limited value.

4.2.2 Gilb's Evolutionary Model

Gilb (1988) considers using a 'quality template' rather than a rigid hierarchical model (figure 4.3). The template is designed to be tailored to local requirements. Crucial quality criteria must be identified and the extent to which these must be present is defined. The software then has quality, in terms of these critical resources, explicitly built into it. Evolutionary development is seen as being critical to the satisfaction of these critical criteria. This is based on the argument that trying to specify a system at the start of a project is difficult and time-consuming. By actually developing part of the system the developer is moving closer to the actual goal, gaining an understanding of the needs of the user, finding errors earlier and providing the user with a usable deliverable at an early stage.

Gilb's template is based on four quality attributes :

1. **Workability**: the ability of the system to do work. This encompasses process
   capacity, storage capacity and responsiveness.
2. **Availability**: the proportion of elapsed time during which a system is able to be used. This incorporates criteria such as reliability, maintainability and integrity.
3. **Adaptability**: this may be considered in terms of improvability, extendability and portability.

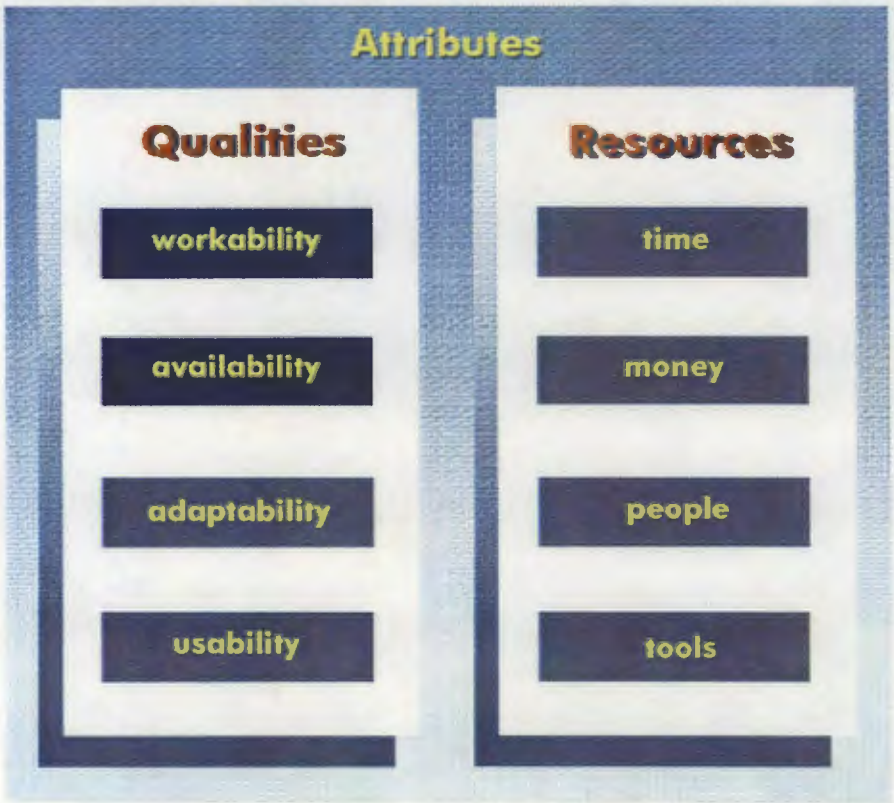

Fig. 4.3 Gilb's Quality Template

4. Usability: the ease and effectiveness of use of a system.
He also highlights four resource attributes : time, money, people and tools.

4.2.3 The COQUAMO project

Kitchenham (1989) introduces the concept of a 'quality profile', making a distinction between subjective and objective measures of quality.

- Transcendent properties are qualitative factors which are hard to measure and about which people have different views and definitions.
- Quality factors are characteristics of the system which are made up of quality metrics and quality attributes. These quality factors are themselves either subjective or objective.
- Merit Indices subjectively define functions of the system. They are measured by quality ratings.

This work has led to a model known as COQUAMO (Constructive Quality Model) which aims to :

a. Predict final product quality
b. Monitor progress towards a quality product
c. Feed back the results to improve predictions for the next project.

4.2.4 The Japanese Perspective

In the early 1950s, the Japanese electronics industry faced a grim reality: their products were not selling. The problem was in the quality of the product rather than the price so they directed their efforts at improving quality.
Quality control in Japan emphasizes the following aspects :

- Quality must be the highest priority.
- All personnel must be involved.
- Quality control must be oriented toward the consumer.

The process of quality assurance is built around three key stages :

a. Design review and document inspection.
b. Intermediate quality Audit.
c. Product and system inspection.

However, although this approach intends to ensure that software is error-free this is still not enough. It is also important to ensure that the system addresses the right question as well as ensuring that it provides a good answer.
Kaoru Ishikawa (Ishikawa, 1985) names six features of quality work in Japan :

- Company-wide quality control.
- Top management quality control audit.
- Industrial education and training.
- Quality circles activities.

- Application of statistical methods.
- Nationwide quality control promotion activities.

4.2.5 LOQUM

According to Gillies (1992), the principal problems with existing models are :

- a technical bias, with a corresponding lack of criteria addressing issues relating to satisfying user needs.
- a lack of good measures.
- a lack of an overall view of quality.
- a failure to address communication issues arising between developers and clients.
- insufficient guidance to make use of models such as Gilb's quality templates.

Therefore he proposes a simple set of tools and procedures to support the practitioner in the construction of his own model, tailored to his own situation. The procedures, known collectively as LOQUM (LOcally defined Quality Modelling), consist of three main stages with a degree of analysis required between each.

These are:

LOCRIT : elicitation of quality Criteria and measures

LOCREL : elicitation of RELationships between quality criteria.

LOCPRO: LOCally defined quality PROfiles.

Each of these stages is reviewed next.

4.2.5.1 LOCRIT

LOCRIT is a knowledge elicitation exercise to derive the relevant quality criteria and associated measures. A group of three people and an enabler should be gathered together within the organisation. The group should preferably consist of a non-specialist user of IT, a Project Manager and a representative of the developer camp. The group are presented with a list of quality criteria and associated definitions which is to act as a prototype, to be refined until it reflects the consensus view. Before any discussion begins, each participant should write down what he considers to be omitted or incorrect within the prototype model. Once the criteria are established, a copy of the final list should be given to each participant and the group should then be asked how they propose to measure the criteria.

4.2.5.2 LOCREL

LOCREL consists of further knowledge elicitation to define relationships and conflicts between criteria. The same personnel as in LOCRIT should participate in LOCREL. Relationships between the quality criteria are classified as direct, inverse or neutral. The participants are presented with a grid of relationships and asked to classify each into one of the above three categories.

4.2.5.3 LOCPRO

LOCPRO is a profiling tool to display a graphical profile to represent the overall quality of a system. A circular graph, or profile, based on the scores assigned to the performance indicated outlined in LOCRIT is plotted. These profiles can then be compared against an 'ideal' template, or against other profiles. A number of factors as well as the values of the quality measures will influence the area contained by a profile of this type but a rigorous discussion of these is outside the scope of this work.

4.2.6 Quality Assurance as a Measurement Science

This technique is based on work done by Schulmeyer et al (1987). The basic method to be utilized in identifying the important quality factors is a software quality requirements form. This form lists 11 quality factors each of which the client is requested to mark as either VI (Very Important), I (Important), SI (Somewhat Important) or NI (Not Important) to him.

He is then asked some questions in order to assess his standing in the software development team organisation and finally he is requested to consider the basic characteristics of the application under discussion. These basic characteristics will have a bearing on the list of quality factors produced by this method as software quality requirements can be strongly influenced by characteristics such as the fact that this is a real-time application, or that human lives are affected. A tentative list of quality factors will be produced and the interrelationships between these should be considered as this will also have a bearing on the final scenario.

The form is described below in detail.

Figure 4.4

Software Requirements Survey Form

1. The 11 quality factors listed below have been isolated from the current literature. They are not meant to be exhaustive, but to reflect what is currently thought to be important. Please indicate whether you consider each factor to be Very Important (VI), Important (I), Somewhat Important (SI), or Not Important (NI) as design goals in the system you are currently working on.

The 11 quality factors are:

Correctness	The extent to which a program satisfies its specifications and fulfils the user's mission objectives.
Reliability	Extent to which a program can be expected to perform its intended function with required precision.
Efficiency	The amount of computing resources and code required by a program to perform a function.
Integrity	Extent to which access to software or data by unauthorised persons can be controlled.
Usability	Effort required to learn, operate, prepare input, and interpret output of a system.
Maintainability	Effort required to locate and fix an error in an operational program.
Testability	Effort required to test a program to ensure it performs its intended function.
Flexibility	Effort required to modify an operational program.
Portability	Effort required to transfer a program from one hardware configuration and/or software environment to another.
Reusability	Extent to which a program can be used in other applications.
Interoperability	Effort required to couple one system with another.

2. What type(s) of application are you currently involved in?
3. Are you currently in:
   1. Development phase
   2. Operations/Maintenance phase
4. Please indicate the title which most closely describes your position:
   1. Program manager
   2. Technical Consultant
   3. Systems Analyst
   4. Other (please specify)

To complete the survey, the following procedure should be followed:

Consider basic characteristics of the application.

The software quality requirements for each system are unique and are influenced by system or application-dependent characteristics. There are basic characteristics which affect the quality requirements: therefore each system must be evaluated for its basic characteristics:

CHARACTERISTIC	QUALITY FACTOR
- If human lives are affected	Reliability
	Correctness
	Testability
- Long life cycle	Maintainability
	Flexibility
	Portability
- Real time application	Efficiency
	Reliability
	Correctness
- On board computer application	Efficiency
	Reliability
	Correctness
- Processes classified information	Integrity
- Interrelated systems	Interoperability

Once a tentative list of quality factors is produced, the interrelationships between the selected factors must be considered. Some factors are synergistic, while others conflict. The impact of conflicting factors is the cost to implement will increase, lowering the benefit-to-cost ratio.

Definitions of quality criteria should be provided as part of the software specification.

4.2.7 Quality Programming

Quality programming is based directly on the waterfall life cycle model of development. Each stage of the life cycle has quality considerations associated with it.

4.2.7.1 Modelling

Given a system to be developed, a model is developed to analyse and understand the problem. The modelling activities may take several iterations to thoroughly understand the problem. The following are the major tasks that must be done during this phase :

1. An exact, unambiguous and complete problem description is essential.
2. The input variables and their source must be completely and exactly defined.
3. The characteristics of the input variables should be analysed e.g. upper and lower bounds of numerical variables, components of compound variables etc.
4. Rules governing the use of data must be defined.
5. The definition of the expected output of the system.
6. The definition of minimum quality standards that the system output is required to meet.
7. Different methods may be used to solve a problem in software development. As many of these methods as possible should be studied and documented for later use.
8. The characteristics of a piece of software, such as the output data generation capacity, speed, memory & data storage requirements, data flows and the like must be studied at this stage of the software development process.
9. Questions as to how the software should be developed should be asked. Should the software be developed from scratch? Are there existing software packages available? Should a software prototype be developed first? And so on.

4.2.7.2 Requirements Specification

Requirements are then generated as a result of the modelling activity. Included in the requirements are software and test requirements. Software requirements define the functions the software is to perform and the quality characteristics such as response time, understandability and portability. Test requirements are specified as the criteria for software testing and acceptance upon completion. They consist of the definition of the expected system output, definition of minimum quality standards, software acceptance criteria, sampling methods and software reliability confidence level.

4.2.7.3 Concurrent Software Design and Test Design

With well-defined requirements, software development can be divided into two channels which can proceed concurrently: software design and implementation and software test design and implementation. Top-down design, structured programming and critical-module-first implementation methods are used in the software channel.

There are many software design methods in practice. These may be classified into two groups: function-oriented design and object-oriented design. The design process will proceed differently according to which type of design method is used.

The formulation of sampling plans for estimating software defect rate and population acceptability, the design of input units, and the implementation of the sampling plans are used in the test channel.

During the design and implementation phases, interfaces between the channels are incorporated to ensure that quality is built into the software at every stage of the development. The two channels meet at the time of testing and integration. If the software passes the test satisfying the test requirements, delivery to the user takes place.

4.2.7.4 Software Implementation

Software implementation consists of two major tasks: coding and test case generation with expected results of processing the input.

4.2.7.5 Testing and Integration

After software design and test design have been implemented, the individual models are to be tested before being integrated into the software. Testing and integration should be done 'bottom-up'. After all the modules have been tested and integrated, the software is then tested as a whole.

4.2.7.6 Software Acceptance

If the developed software satisfies the quality requirements it is delivered to the user. If the previous stages have been followed rigorously, an acceptable system should always be the result.

4.3 Quality Assurance Survey

Quality assurance has become a rapidly growing part of data processing. In 1989, the Quality Assurance Institute completed a survey of its members. The significant findings of this survey as reported by Schulmeyer et al (1989) were:

1. The primary focus of the QA groups represented was on standards and procedures development and improvement.
2. The major impediment facing QA groups was the inability to obtain management's commitment and involvement.
3. Many organizations have been able to prove the value of quality and quality's positive relationship to productivity.
4. Measurement was considered to be one of the most important new activities on which the QA groups were focusing.
5. QA's view of its own mission was shifting from a passive role focused on corrective action to a leadership role focused on process management.

The survey conclusions were:

1. Members of the QA function need to develop strong skills in the areas of communication and marketing if they wish to overcome the continuing problem of lack of management commitment.
2. Measurement data is needed to prove the value of quality in order to gain management's long term commitment to quality.
3. The mission of the QA function must move toward one of leadership and one which will lead the information services organization into a culture of continuous process improvement.

Using the survey findings and conclusions, the Quality Assurance Institute recommends the following two actions for QA managers:

1. Position QA as a leadership function by establishing a vision and then developing programs to accomplish that vision. QA managers should begin this process by establishing small short term visions that can be accomplished with existing staff. Success builds credibility.
2. Demonstrate that quality works by selecting one or two of the results desired by MIS managers which can be aided by quality programs. Develop a measurement program and then benchmark the current status for those results. Present a plan to move the results in the desired direction and make it happen.

4.4 Summary and Conclusions

Different models and methods for attaining quality in software have been examined in this chapter.

As Glass (1992) points out, there are currently no 'best' approaches to achieving software quality. Thus, approaches to software quality must depend on several factors : The application problem being solved, the organisation solving the problem and the people within the organisation solving the problem.

CHAPTER 5

PROPOSED QUALITY ASSURANCE REFERENCE MODEL

CHAPTER 5

PROPOSED QUALITY ASSURANCE REFERENCE MODEL

5.1 Introduction

The proposed quality assurance reference model for object orientation closely follows the pattern of the Spiral Model for object-oriented development and will therefore be referred to as the Spiral Quality Assurance Reference Model for Object-Orientation or SQARMOO. Although the point of departure for this study is object-orientation, this model could equally be adapted to other development processes. In terms of this model, quality assurance can be viewed at three levels: Universal, Worldly and Atomic. The Universal Level provides a global view of the whole quality assurance process taken by top level management. Middle management is concerned with a more detailed approach to quality assurance at the Worldly level while junior management is concerned with the minutiae of achieving, measuring and controlling quality at the Atomic level.

Level	Domain	QA Tasks
Universal	Complete Spiral Model	Management-oriented view of system quality
Worldly	A particular Spiral	Relate managerial view to technical aspect
Atomic	A quadrant of a particular cycle	Technical view of system quality

Table 5.1 Methodology for Establishing QA for IS Development

The revised Spiral Life Cycle Model has software development as an iterative process, taking place over 5 cycles. This iterative development procedure maps cleanly onto the quality assurance process as well. During Cycle 1 the feasibility of the project as a whole and of the ability to enforce quality standards in particular, is reviewed. Cycle 2 is concerned with the architecture of the proposed system and its effect on the overall quality of the system. During Cycle 3, the Analysis cycle, the criteria which are applicable to the project and their target values are identified, if possible. Cycle 4 deals with the design of quality assurance techniques, and identification of metrics. Finally, during Cycle 5 the techniques and methods identified in the previous four cycles are applied and evaluated.

Finally, each cycle is divided into 4 quadrants. The first Quadrant deals with the formulation of quality assurance criteria and standards in terms of the current cycle. During Quadrant 2, metrics, testing tools and other quality assurance techniques are analysed for use in the measurement of the quality criteria. In Quadrant 3, a quality assurance plan is developed using the techniques identified in Quadrant 2 to measure whether the standards formulated in Quadrant 1 are being met, while in Quadrant 4 the results of the quality assurance plan are reviewed and a quality assurance report is produced.

5.2 Software Quality Levels

Software quality can be measured at 3 levels (figure 5.1). These are discussed below.

5.2.1 Level 1 - Universal

The software quality metrics framework begins with quality criteria which represent the management-oriented view of system quality. At this level, the quality criteria that are important for the project are established by top management. Priorities and weighting factors are also allocated to the various criteria at the Universal level. Associated with each criterion are one or more quality factors and, if possible, a target value. For example, management may consider the maintainability of the system very important with a target 'value' being that 'no corrective action should take more than 48 hours to implement'. Budget and time constraints are also identified at this level as are resources, both human and machine. The output from this level is a global quality assurance plan defining the quality criteria and their target values where applicable, a budget, target dates for the various phases of the project and resource allocations.

5.2.2 Level 2 - Worldly

At the Worldly Level we identify quality factors, which are a bridge between the managerial and the technically-oriented views of system quality. These are obtained by decomposing each quality criterion into measurable software attributes. Quality factors are independent attributes of software, and therefore may correspond to more than one criterion. Maintainability could, for example, be decomposed into the factors 'correctability', 'testability' and 'expandability'. If a target value was associated with the criterion at level 1, this may be inherited by the factors. Otherwise, an attempt should be made to associate sub-target values with each factor at this level.

It is at the Worldly level that the Quality Assurance team should be formed.

5.2.3 Level 3 - Atomic

The Atomic level deals with the technical minutiae of quality assurance. The quality criteria and quality factors discussed at the previous 2 levels are decomposed into metrics used to measure system products and processes during the development life cycle.

The metrics used to finally measure the maintainability of a system could be any (or all) of the following :

Fault counts	Closure time
	Isolate/fix time
	Fault rate
Degree of testing	Statement coverage
	Branch coverage
	Test plan completeness
Effort	Resource prediction
	Effort expenditure
Change counts	Change effort
	Change size
	Change rate

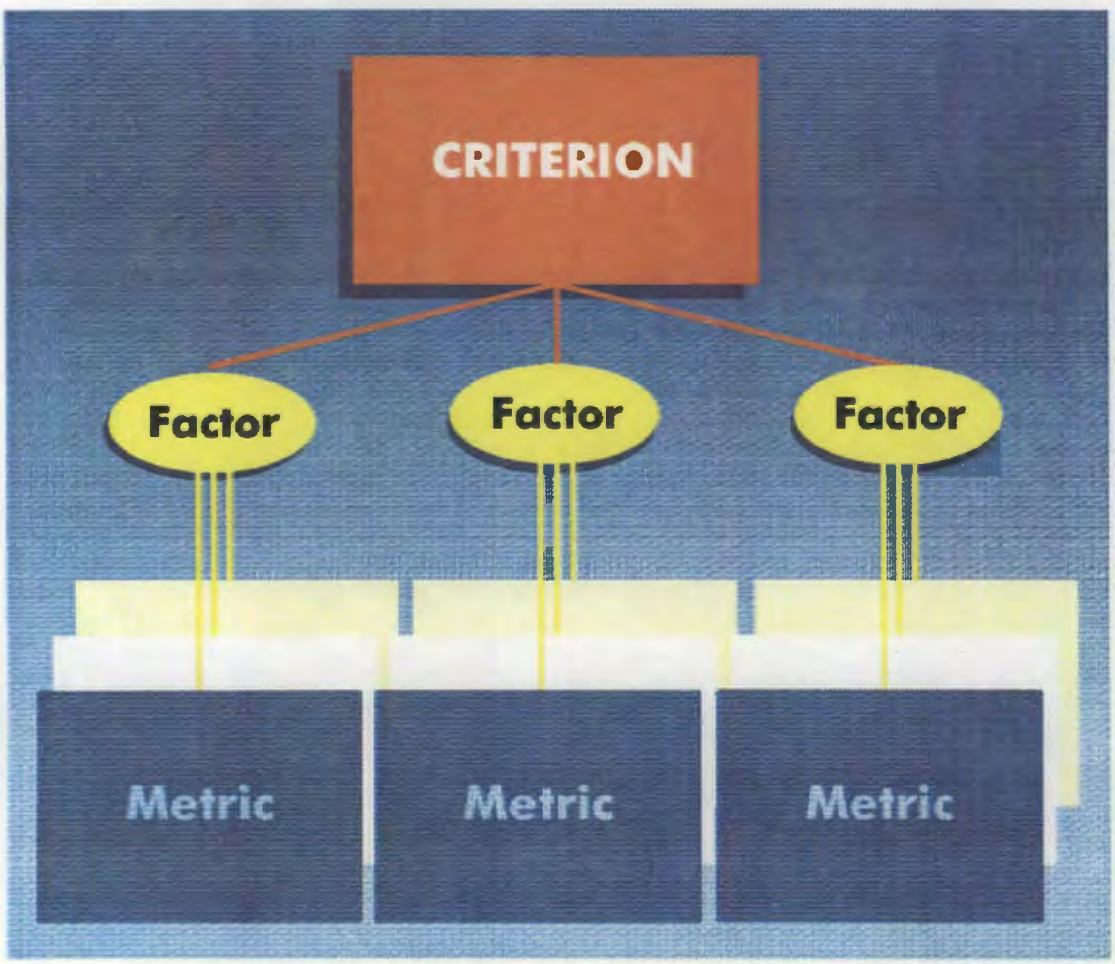

Fig. 5.1 **A Software Quality Measurement Framework**

Quality Criteria	Criterion 1 C_1	Criterion 2 C_2	...	Criterion n C_n
Target Values	Target Value 1 TV_1	Target Value 2 TV_2	... ...	Target Value n TV_n
$Q = \{ C_1TV_1, C_2TV_2, ..., C_nTV_n \}$				

Table 5.3 Universal Level

NOTE : It may not always be possible to associate target values to criteria at this level.

Quality Criteria	Criterion 1 C_1			
Factors	Factor 1 F_1	Factor 2 F_2	... ...	Factor n F_n
Target Value	Target Value TV_1			
Sub-Values	Sub-Value 1 STV_1	Sub-Value 2 STV_2	... ...	Sub-Value n STV_n
$C_1 = \{ F_1, F_2, ..., F_n \}$ $TV_1 = \{STV_1, STV_2, ..., STV_n\}$				

Table 5.4 Worldly Level

NOTE : It may not always be possible to associate sub-target values to factors at this level.

Quality Factors	Factor 1 F_1			
Metrics	Metric 1 M_1	Metric 2 M_2	... ...	Metric n M_n
Sub-Values	Sub-Value 1 STV_1			
Metric Values	Metric Value 1 MV_1	Metric Value 2 MV_2	... ...	Metric Value n MV_n
$F1 = \{ M_1, M_2, ..., M_n \}$				
$STV1 = \{ MV_1, MV_2, ..., MV_n \}$				

Table 5.5 Atomic Level

NOTE : Some form of Metric Value MUST be associated with each metric at this level or else there can be no formal measure of quality achieved.

5.3 Specification of a QA Methodology for the Complete Revised Spiral Model

A matrix of Quality Assurance Tasks can be mapped against the Revised Spiral Life Cycle Model, with each task being associated with a particular quadrant within a particular cycle (figure 5.2).

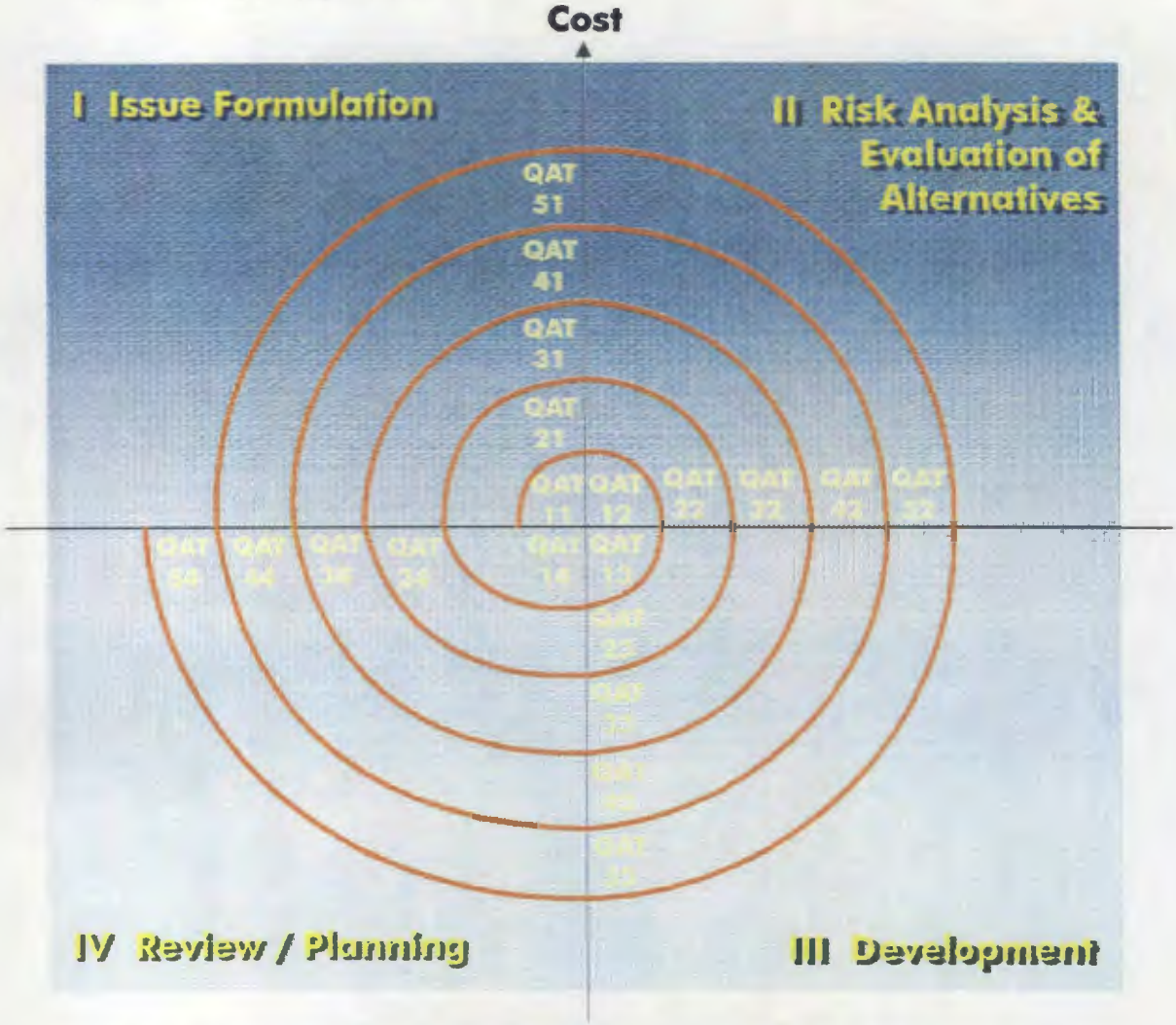

Fig. 5.2 **The Spiral Quality Assurance Reference Model for Object-Orientation**

	QUADRANTS			
Cycles	1	2	3	4
1	QAT11	QAT12	QAT13	QAT14
2	QAT21	QAT22	QAT23	QAT24
3	QAT31	QAT32	QAT33	QAT34
4	QAT41	QAT42	QAT43	QAT44
5	QAT51	QAT52	QAT53	QAT54
QAT = Quality Assurance Task				

Table 5.6 Quality Assurance Matrix

QAT11: FEASIBILITY - Issue Formulation

The system requirements must be closely studied in order to establish if the requirements are feasible within the constraints given. It should be ascertained that, from a Universal Level or global viewpoint, in terms of the broad systems requirements, quality is achievable.

QAT12: FEASIBILITY - Analysis

It must be decided whether it will be possible to measure software quality for the system, and if so, identify which quality criteria are applicable. This could be done by presenting the users with a list of criteria and asking them to indicate which criteria they consider important and in what way. Technical criteria will have to be decided on by the software engineering department, or in terms of existing software standards.

QAT13: FEASIBILITY - Development.

Each of the quality criteria identified as being important to the overall quality of the system should be given a score indicating its perceived importance within the system. To allocate these scores, organisational experience and required standards and regulations should be used. It should be established whether or not it would be possible to set up metrics for these requirements and it must be determined how they are going to be measured.

QAT14: FEASIBILITY - Review/Planning

The scores allocated by all involved parties should be surveyed and a final, agreed upon list of priorities created. An initial Software Quality Requirements Plan should be drawn up listing the criteria identified, their relative degrees of importance and any target values associated with them.

QAT21: ARCHITECTURE - Issue Formulation

Once the system architecture has been decided, the criteria identified in cycle 1 must be re-examined to decide whether or not they are still feasible. For example, it may no longer be feasible to measure such aspects as understandability or modularity if it has been decided that the system will have to be developed in assembler language.

QAT22: ARCHITECTURE - Analysis

Analysis must be done on each criterion to see how the system architecture affects it. For example, the choice of a particular operating system or programming language may affect the portability of the software. A choice of programming language can affect many aspects like maintainability, efficiency,and understandability, while a choice of computer system could influence the efficiency of the system.

It is also the responsibility of the Quality Assurance Managers to provide input to the decision making process on system architecture. They should be allowed to influence the choice made based on of quality considerations as they should be well-versed in the effects on quality of various architectural options.

QAT23: ARCHITECTURE - Development.

Once the proposed system architecture has been decided, its affect on the system quality should be analysed.

It may be necessary to establish new quality criteria or assign different weights to criteria based on this new information.

QAT24: ARCHITECTURE - Review/Planning

Review the amended list of criteria and revise the Software Quality Assurance Plan accordingly. Decisions made about various architectural aspects of the system that may impact on the final quality of the system should be detailed in this plan. For example, it may be necessary to explain why a particular operating environment, or programming language was chosen, if quality considerations were influential in its choice.

QAT31: ANALYSIS - Issue Formulation

Once analysis of the system is complete, a more thorough understanding of how it is to operate should be possible. Based on this, new quality issues may come to the fore, for example, the need for tighter security or more rigorous integrity checking may be identified. Also at this point, the OMT methodology comes into play with the initial object, dynamic and functional models being designed.

QAT32: ANALYSIS - Analysis

The object, dynamic and functional models should be thoroughly analysed by the QA group to ensure that they are in line with system and quality requirements.

QAT33: ANALYSIS - Development

The object, dynamic and functional models will be further enhanced during this phase. The quality assurance task at this juncture is to ensure that they are developed in line with the models proposed in QAT32 and that quality aspects are not compromised.

QAT34: ANALYSIS - Review/Planning

A formal examination of the three models developed up to this point should be performed in order to verify that the functional analysis is a correct expansion of the program performance specification and the Software Quality Assurance Plan. Detailed documentation on the ways in which these models were tested (walkthroughs, inspections, etc) should be kept. The SQAP will be revised at this point to include this documentation.

QAT41: DESIGN - Issue Formulation

Once a system has been designed, the level of coupling and cohesion between modules (or objects) can be measured. As Fenton (1991) says

"The only way to evaluate a design is by examining the volume and complexity of the interfaces, specifically the data interfaces. If this kind of evaluation has not been performed, there is no reason for believing the design."

As the object-oriented paradigm is based on data abstraction, there should not be a problem here, nevertheless, if the objects are not well-designed, an inspection of this nature might highlight the difficulties.

QAT42: DESIGN - Analysis

Metrics to determine the cohesion within modules/objects and the coupling between them should be applied to the system.

According to Fenton (1991), there is no obvious measurement procedure for determining the level of cohesion in a given module, but it should be possible to describe the module's function in a single sentence. If this is not the case, then the module is not likely to be functionally cohesive and may need to be redesigned. One measure for the level of coupling between two modules, x and y, as given by Fenton (1991) is :

$$c(x,y) = i + n/(n+1)$$

where i is the number corresponding to the worst coupling type between x and y (taken from the scale given below) and n is the number of interconnections between x and y.

Coupling Types are:

Content Coupling	5
Common Coupling	4
Control Coupling	3
Stamp Coupling	2
Data Coupling	1
No Coupling	0

A low value for c(x,y) is desirable. It may be necessary to request the redevelopment of certain objects if they do not meet the required standards here.

QAT43: DESIGN - Development.

Apply the measures described above, or similar ones, to the system to determine the level of cohesion and coupling. Other key objectives to be considered here are:

- Accuracy with high performance
- Reliability and fault tolerance
- Flexibility to accommodate change and growth

- Testability
- Maintainability

The system should also be evaluated fully with respect to all the quality criteria mentioned in the SQAP.

QAT44: DESIGN - Review/Planning

A detailed design inspection should be made at this stage. The software development team will want to stepwise refine the high-level design to an intermediate level before translation to the target language code can be authorised. Thus, it is the QA team's task to check that the high-level design is still in line with the initial requirements documents and the SQAP. All interfaces between processes, tasks and objects should be checked for completeness, correctness and consistency.

QAT51: IMPLEMENTATION - Issue Formulation

For each metric in the metric set defined in the SQAP, determine the data that must be collected and any assumptions made about the data.

The flow of data should be shown from point of collection to evaluation of metrics. Describe when and how tools are to be used, identify the organisational entities that will directly participate in data collection and describe the training required for each metric. Different departments will measure the characteristics of the system and compare them with the functional specification and the quality requirements. These departments should be aware that the measurement of quality characteristics requires extra effort and should have budgeted their time accordingly.

QAT52: IMPLEMENTATION - Analysis

Test the data collection and metric computation procedures on selected software. Determine the cost of this prototype effort to further refine the cost estimates and select the appropriate set of tools (manual or automated) to satisfy the requirements for data collection and metrics computations. Collect and store the data at the appropriate time in and compute the metric values from the collected data.

Different types of tests should include:

- The functionality and general efficiency of the system as well as the other user-criteria as specified in the SQAP should be evaluated by the potential users of the system.
- The computer centre should examine the machine efficiency of the system.
- Technical aspects of quality e.g. flexibility, maintainablity and so on, should be examined by suitably quality members of the SQA team.

- The internal accountant should check the correctness, completeness, authorization and timeliness of the information provision function.

A detailed code inspection should be performed. The code inspection serves to verify that the code performs to the specified requirements. It should also be checked for conformance to company standards e.g. modules should not exceed the maximum agreed upon length, meaningful or company-standard variable names should have been used, indentation should be to the agreed upon standard and so on. This is also a vehicle for the early audit of the code by the programer's peers and for the early detection of errors. The module interface requirements should be verified as should the modules test specifications.

QAT53: IMPLEMENTATION - ~~Development.~~

The results should be interpreted and recorded against the broad context of the project as well as for a particular product or process of the project. Identify software components which appear to be of unacceptable quality. Use already validated metrics to make predictions of direct metric values and compare these to target values to determine whether to flag software components for detailed analysis.

QAT54: IMPLEMENTATION - Review/Planning

A final Software Quality Assurance Document should be produced detailing all the quality assurance work performed.

It should include the revised SQAP, as well as the evaluations performed on the Object, Dynamic and Functional Models. The entire quality data gathering phase carried out under QAT52 should be thoroughly documented as should any decisions taken, conclusions reached or predictions made during QAT53.

5.4 Summary and Conclusions

A Spiral Quality Assurance Reference Model for Object-Orientation (SQARMOO) is developed in this chapter. User, developer, managerial and technical definitions of what constitutes quality are taken into consideration. Object-Orientation and the Revised Spiral Life Cycle Model form the QA methodology with the Spiral Model being mapped onto a matrix of Quality Assurance Tasks. Each cycle in the Spiral Model constitutes a row in the QA matrix, while each column of the matrix represents a quadrant from the Spiral Model. This matrix could be extended along a third dimension to represent the various levels as discussed in Section 5.2, but this has not been done here. Finally, the Software Quality Measurement Framework outlined in Section 5.2 is applied. This model maps directly onto the Quality triangle discussed in Section 1.2 thus answering the questions proposed there (see figure 5.3).

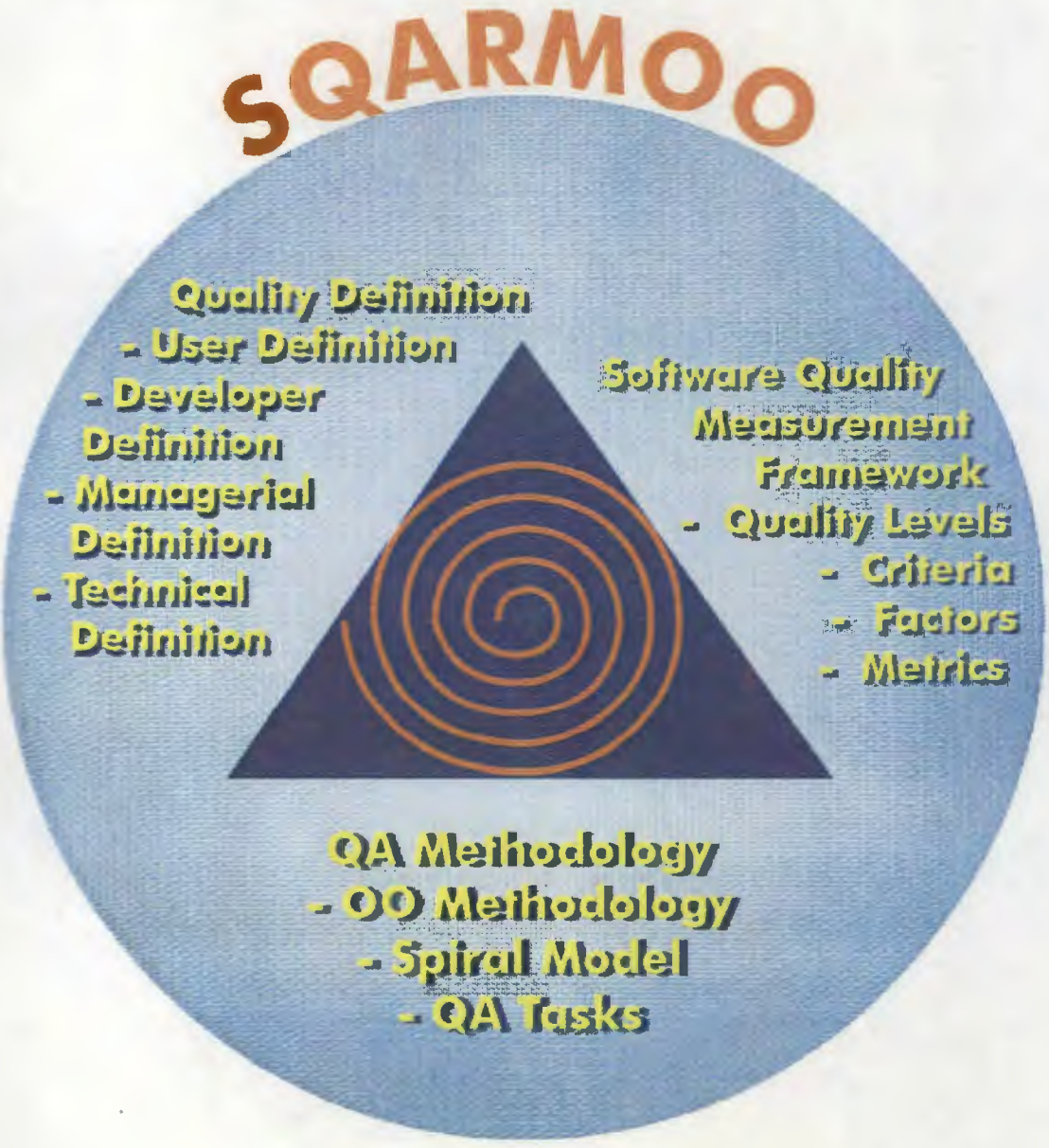

Fig. 5.3 The Quality Triangle - Solutions

CHAPTER 6

DEMONSTRATION OF CONCEPT

CHAPTER 6

DEMONSTRATION OF CONCEPT

6.1 Introduction

This chapter presents a scenario where the proposed quality assurance reference model, SQARMOO, is applied to a small software project. A more detailed analysis of the model is outside the scope of this dissertation, but this scenario should, at least, serve to illustrate the major concepts behind the model.

6.2 Project Background Information

At Duvha Power Station, one of Eskom's coal burning stations, there are six units each producing 600 MegaWatts. There is a control room for each unit from which an operator can make adjustments to various parameters affecting the operation of the unit. Occasionally, a change to some item in the plant may cause a Temporary Operating Instruction (TOI) to be issued. These instructions are valid for a maximum period of 30 days. It is vital that all the operators have a list of the TOIs currently valid for their units always easily accessible. It was therefore decided that it is necessary to enable the operators to keep these instructions electronically on the computer instead of in written format as was previously the case. Expired TOIs should remain somewhere on the system as they are legally binding documents which may be needed to settle disputes in the future.

6.3 Project Requirements

A computerised system is needed into which Temporary Operating Instructions (TOIs) may be entered and later viewed. TOIs will be entered by only a few authorised personnel. No other person should be able to enter a TOI.
There must be a facility whereby a TOI can be cancelled, but TOIs may not be modified or deleted once they have been accepted into the system.
An operator must be able to easily locate and view all TOIs applicable to his unit quickly and reliably. An ordinary document retrieval system is not adequate because the document titles are usually obscure and it is difficult for an operator to identify the document he needs.
Once a TOI has expired it should be kept on the system, but there should be some noticeable indication that it has, in fact, expired.
The system must be available 24 hours a day, including Sundays and must always reflect up-to-date information. It should be easy and efficient to use.

6.4 Definition of Quality

In their document EVS 010, Eskom adopts the following definition of quality :
"Quality is the conformance to requirements, driven by a system of error prevention with the performance standard of zero deviation from requirements and the measure of quality as the price of non-conformance."
Thus, Eskom places a high value on the conformance to requirements when ascertaining quality.

6.5 Applying SQARMOO to the TOI Program

QA11: FEASIBILITY - Issue Formulation

As this is a small uncomplicated system, there should be no difficulty in achieving a high level of quality. Two meetings with the users of the system should be adequate to refine the requirements to the point where a more detailed analysis and design of the system can proceed.

QA12: FEASIBILITY - Analysis

The users and software developers were presented with the list of quality criteria detailed in Section 2.2.3.2. The criteria which they identified as important to the project are the following:

User Quality Criteria

1. User friendliness — the system must be easy to learn and easy to use as most currently employed unit operators have little previous experience of working on a computer.
2. Response Time — information should be displayed after a maximum of 5 seconds delay.
3. Reliability — it is imperative that the system be available to the operator at all times, network failure, for example, cannot be allowed to interfere with the retrieval of TOIs. In addition, the information dis played to the operator must be correct. Invalid information could have disastrous results.
4. Robustness — the system should be able to withstand incorrect usage or conditions without crashing out.
5. Security — TOIs are legally binding documents. It must not be possible for these instructions to be tampered with, or for TOIs to be added to or removed from the system by unauthorised personnel.
6. Completeness — the system must be able to fulfil all the needs of the operator, not just a subset of them.

Technical Quality Criteria

1. Complexity	the system should not be unnecessarily complex.
2. Understandability	the system design should be clear and easy to understand.
3. Maintainability	as Eskom will be taking over the maintenance of the system, this should be easy to do.
4. Reusability	the extent to which this product will satisfy the needs of the other power stations.
5. Efficiency	the speed of the system is of highest concern with disk and CPU usage being of secondary concern.
6. Testability	because of the critical nature of the information involved, it is crucial that the system be thoroughly tested before being delivered, and once in place at the power station, it must run in parallel to the current manual system until it has proved reliable.
7. Modifiability	it may be necessary to add more information to the system at a later stage.
8. Interoperability	it is vitally important that the TOI system integrates seamlessly into the Eskom environment.

QA13: FEASIBILITY - Development

The quality criteria were then given one of three possible ratings: Critical, Very Important and Important as detailed below:

CRITICAL

Security
Reliability
Robustness
Completeness

VERY IMPORTANT

User friendliness
Response Time
Maintainability
Reusability
Efficiency

IMPORTANT

Complexity
Understandability
Testability
Modifiability
Interoperability

QAT14: FEASIBILITY - Review/Planning

INITIAL QUALITY ASSURANCE REQUIREMENTS PLAN FOR DUVHA POWER STATION TEMPORARY OPERATING INSTRUCTION (TOI) PROGRAM

The operating and data processing staff at Duvha Power Station have identified the following criteria as being either Critical, Very Important or Important with regard to the development of the Temporary Operating Instruction (TOI) system.

CRITICAL

Security : Capture of TOIs to the system must be done only after the necessary paperwork has been compiled and authorised and should be strictly controlled. The following recommendations are made to ensure the security of the system.

1. Input to the system should be password protected. The pass word should be known only by the Secretary of the Operating Manager and the Senior Shift Supervisors and should be changed weekly or whenever it is suspected that password security has been violated. Thus, there should be a facility on the system whereby the password for the capture of TOIs may be changed. The system should require the entry of this password every time the user wishes to add a new TOI to the system. The password will be stored in a database file and it will be encrypted.
2. It will not be possible for the user to delete or modify any TOIs entered into the system. Thus, it will be possible to see if any body has attempted to 'play' with the system.
3. All TOI database files and related documents will be marked with read-only access for everybody except for those people authorised to enter TOIs.

Reliability : The system must be available 24 hours a day, 7 days a week and should provide up-to-date information.

Robustness: The system must be able to stand up to a fair amount of abuse. It should prevent the possible entry of incorrect TOI numbers by generating them automatically. The numbers will take the form 'TOInnnnn' where nnnnn is an automatically generated numeric starting at 00001. The system must ensure that the user is not able to enter TOIs that are valid for more than 30 days, and that an AKZ number is always entered. It must check all dates and times for validity. Mistakes made by the user on entering a TOI should be able to be rectified up until the time the TOI is accepted into the system.

Completeness: The program must provide a complete TOI function for the operators. It must not be necessary for them to have to access information on TOIs from any other system or person.

VERY IMPORTANT

User Friendliness : It should not be necessary to provide extensive training in order for someone to operate this program. All operations should be clearly demarcated in some way and there should be adequate on-line help. It should not take more than a 2 hour demonstration and a 2 hour hands-on period for an operator to become familiar with the workings of the system.

Response Time: Once the user asks to view a particular TOI the system should respond in no more than 5 seconds. User input should be limited only by the typing speed of the user.

Maintainability: It should not be necessary to rewrite the whole system in order to add a new button to the screen, or an additional field to the database.

Reusability : The product should be written in such a way that it could be put into use at another power station with the minimum of effort.

Efficiency : This is a small program. The executable file should not occupy more than 1MB of disk space and should require less than xxxKB of RAM in order to execute.

IMPORTANT

Complexity : The computer program should, as closely as possible, mirror the manual procedures followed at present. This is a simple process and it should not be unduly complicated by computerizing it. The system should per form its function neatly and in an attractive way. It is not expected to provide any additional frills such as videos, animation or sounds.

Understandability : It should be possible for an experienced programmer to gain a reasonably clear understanding of the way the system has been set up within 5 days of looking at the source code and documentation.

Testability : This is not a complicated system and it should be possible to test it thoroughly simply by getting people of different capabilities to interact with the system for a while, entering, viewing and cancelling TOIs.

Modifiability: The system should be open-ended enough that additions and enhancements can be made later without causing any problems. Database design should be such that modifications to the tables at a later stage will not cause any problems.

Interoperability :The program must comply with the Eskom Project R guidelines and should integrate with Eskom's IGIS project.

QAT21: ARCHITECTURE - Issue Formulation

Hardware

Duvha has an HP mini computer to which a number of 386 PCs are connected via an Ethernet LAN. The PCs have SVGA 14 inch monitors, 8MB RAM and hard disks varying in size up to 2 GigaBytes. The TOI system would be required to run on this equipment.

Software

Eskom wishes the program to run in a MicroSoft Windows environment. Thus, it was decided that MicroSoft Visual C++ for windows should be the programming language used with the Watcom library providing the database functionality. The instructions are documents and can be several pages long so it was decided that MSWord for Windows should be used for these.

QAT22: ARCHITECTURE - Analysis

Windows is a new operating system at Duvha and the operators, who are not computer literate on the whole, should be screened as much as possible from its functionality. All windows used should be modal so that the operator can't accidently 'click' his way out of them and the maximize and minimize buttons should not be made available so that the operator can't accidently minimize his window.

MicroSoft Visual C++ is not a simple language. It will, therefore, be in the interests of reducing the complexity and enhancing the understandability of the system if naming conventions, indentation conventions and other standards be adhered to. As many as possible of the provided Foundation Class Libraries should be made use of.

As Duvha has already experienced difficulties on their LAN, we cannot rely on information on the UNIX computer being available to the operators 24 hours a day. Therefore, it is proposed that instead of the information being obtained directly from the UNIX computer, a system of replication be applied. The UNIX computer will be regarded as the Replication Master and any new TOIs must be entered into the system via the LAN onto the UNIX computer. However, copies of the program, data and all related documents will reside on the local hard drives of each of the operators' PCs in the 6 units. This means that if there should be trouble with the LAN, or on one of the operators' PCs, nobody else need be affected. To ensure that the system is always up-to-date, it will be necessary to download the data and documents to the operator PCs on a daily basis.

The program should also have a 'Download' button, which should activate a procedure which fetches all the latest relevant information from the UNIX computer. Operators should be instructed to activate this button at the start of each shift or whenever they have been told that a new instruction has been added to the system. There must be a facility whereby a drive & directory mapping can be entered into the system, indicating where on the LAN the TOI system is located in order for the download facility to be easily activated.

QAT23: ARCHITECTURE - Development

Because the actual instruction portion of the TOI will be stored as a separate MSWord document, security issues must be revisited. MSWord provides the ability to modify, delete etc documents. It was therefore decided that while MSWord for Windows will be suitable to use for the capture of TOIs, a viewer program (Outside-In), which provides no additional functionality, will be used to display these documents to the operator.
MSWord and Outside-In are separate computer packages which will be launched from the TOI program. This could mean that the 5 second time limit on information display should be re-assessed.

QAT24: ARCHITECTURE - Review/Planning

A revised SQAP should be produced. In this instance, only the revisions are detailed and the Initial SQAP is still needed to provide a full picture. It is probably easier, however, to completely revise the initial SQAP so that there is only one document to deal with later.

REVISED QUALITY ASSURANCE REQUIREMENTS PLAN FOR DUVHA POWER STATION TEMPORARY OPERATING INSTRUCTION (TOI) PROGRAM

The following revisions should be made to the existing QA requirements document:

CRITICAL

Security: TOI header information will be kept in an encrypted database file. Instructions will be entered using the MSWord for Windows package and all documents will be flagged as read-only to all users except those authorised to enter them. An additional level of security will be provided by the purchase of a specific viewer program which will not allow an operator to do anything more than simply view the document. This program, rather than MSWord for Windows will be used to provide the View Instruction function.

Reliability :The master system will reside on the UNIX computer but will be completely replicated to the local hard drive of each operator's PC. A daily download will be included in the normal LAN housekeeping operation. Operators will also be able to download the latest TOIs themselves by means of simply clicking a button on their programs.

Robustness: As much Windows functionality as possible will be hidden from the operator. The operator PCs will have their stiffy drives removed to prevent unauthorised software being used to circumvent system security.

VERY IMPORTANT

Response Time: When an operator, accessing the TOI system residing on the hard drive of his computer, asks to view a particular TOI the system should respond with the TOI header information in no more than 5 seconds. When he clicks on the 'Instruction' button, it should take no longer than 10 seconds to launch the viewer and load the relevant document. For a user accessing the system via the LAN, we should perhaps allow up to three times as long for access as a lot will depend on the number of people currently logged in to the LAN.

Reusability : The TOI procedure is not specific to Duvha. However, each Power Station has its own form for issuing TOIs. By making the form object selfsufficient, this could easily be rewritten for each power station without any other program objects being affected.

Efficiency : Because C++ has been chosen as the development language, if the program is properly written, it should not be possible to improve much on its efficiency.

IMPORTANT

Understandability : Standard naming conventions for all system objects and variables should be used. The standard development environment provided with MicroSoft C++ should be used.

QAT31: ANALYSIS - Issue Formulation

It is at this point that the OMT methodology is used to develop the Object, Functional and Dynamic models for the system. To design these models in detail is out of the scope of this project, which is concerned only with proving the quality of the models. The correctness and quality of these models could be tested in many different ways. For a small system such as the one under discussion it is probably adequate to simply have a small group of people perform one or more walkthroughs on the various models.

QAT32: ANALYSIS - Analysis

The following are guidelines for checking the validity of the three models.

OBJECT MODEL

1. Check that the model does, in fact, reflect the real-world requirements.
2. See whether it is possible to simplify the model in any way.
3. Check that object names are descriptive and unambiguous.
4. Make sure that references to other objects have not be described as attributes but rather as associations.
5. Where possible, decompose multiple associations into binary associations.
6. Check that all associations have been qualified if at all possible.
7. Check that generalizations are not too deeply nested.
8. Make sure that associations have been defined properly. For example, check that one-to-one associations are not really zero-or-one associations.
9. Check that the object model documentation adequately describes the model.

DYNAMIC MODEL

1. Check all state diagrams for consistency on shared events.
2. Attributes shown in state diagrams should be only those which are relevant to the state, all other attributes should be omitted for clarity.
3. Check that the diagrams correctly distinguish activities from actions. Activities occur over a period of time while actions are instantaneous.
4. Check that incoming and outgoing transitions to each state have been correctly identified.

FUNCTIONAL MODEL

1. Check data flow diagrams for correctness.
2. Check that data stores have been correctly used.

QAT33: ANALYSIS - Development

This task expands on QAT32 by monitoring the further development of the three OMT models to ensure that the quality criteria are not compromised.

QAT34: ANALYSIS - Review/Planning

The SQAP will now be amended to describe the three models and how they were checked for validity. For example:

> QUALITY ASSURANCE REQUIREMENTS PLAN FOR DUVHA POWER STATION TEMPORARY OPERATING INSTRUCTION (TOI) PROGRAM

Evaluation of the Object, Dynamic and Functional Models

The Object, Functional and Dynamic Models have been verified for conformance to quality issues. This verification took place in the form of a walkthrough. The two people who designed the model provided two other software professionals with all the documentation relating to the project one week before the date for which the walkthrough was scheduled, enabling them to study the requirements and the models in some detail. The walkthrough was then chaired by one of the developers who took the others through the models, answering any questions that were posed as they came up. The models were accepted with only very minor modifications which are described below:
(any modifications should be described at this point).
Any issues which needed additional explanation by the developers should also be listed here so that those issues will be clear later on in the development process.

QAT41, QAT42, QAT43, QAT44: DESIGN

The 3 models will now be further expanded upon by the system developers and correspondingly further assessed for conformance to quality considerations. The models should now be tested with particular emphasis as to how they measure up to the quality criteria as specified in the SQAP. In this instance, therefore, checking should be done to ensure :

Security	Do the models provide adequate security checks so as to prevent TOIs from being entered by unauthorised personnel and from being tampered with in any way?
Reliability	Is the procedure for downloading TOIs from the Unix computer reliable? Are there any unanticipated conditions that could occur causing the system to fail? Does the system check that the information that is being entered is correct?
Robustness	is the error checking adequate? Are there any conditions which are not catered for which could cause the system to crash?
Completeness	does the model meet all the requirements as laid down in the requirements specification document? Are those requirements met in full?
User Friendliness	Are the objects realistic models of the real world ? Is there adequate user help?
Response Time	Is the model efficient? Are data searches performed correctly? Are there unnecessary data store access operations?

Maintain-ability	Are the models clear and easy to understand? Is the documentation adequate and accessible.
Reusability	Are the classes well-designed and self-sufficient. Are the levels of coupling low and the levels of cohesion high?
Efficiency	Are operations performed in the most efficient manner? Have data stores been well-designed?
Complexity	Are the models unnecessarily complex?
Understandability	Are the models easy to understand?
Testability	How easy is it to test the system based on the models?
Modifiability	Are the models designed in an open-ended manner? Would it be possible to add new classes or modify existing ones.
Interoper-ability	How do the models fit in with Eskom's other existing systems, and any projects currently under way?

QAT51: IMPLEMENTATION - Issue Formulation

The system will be tested by both the development organisation and the users. The users who will be involved will be briefed that this is a test period and they should not become discouraged if problems are encountered but rather they should be pleased that they are helping to identify the problem areas. A period of two weeks was decided on as being sufficient for such a small system, however, if many problems are encountered, this testing phase will be extended.

QAT52: IMPLEMENTATION - Analysis

The data to be collected, the persons involved and the target values for each criterion are as follows:

Security

Metric	The number of successful unauthorised TOI inputs, deletions and modifications made during a one-week testing period.
Tester	This test will be performed by somebody who is familiar with computer systems and the software that has been used to develop this one.
Data	Details of how the unauthorised access was managed must be kept so that preventative measures can be taken.
Target	The desired value should, of course, be zero successful unauthorised access attempts.

Reliability

Metric	The amount of downtime in a one week period as well as the integrity of the TOIs which have been captured to the system during this period.
Tester	The operators on one of the units at Duvha Power Station will be asked to use the system for a one week period. Any TOIs pertaining to that unit will be entered by the Senior Shift Supervisor onto the LAN and the information downloaded to the unit computer. The operators will access these TOIs any number of times during their shifts.

Data The number of minutes that the system was unavailable for any reason during this period, as well as the reasons for the unavailability, should be logged. Any problems with the download procedure should also be noted. The TOIs remaining on the system will be compared with the manually processed paper copies from which they were captured in the first place.

Target There should be no downtime logged for the entire period. Downloading should have caused no problems and the TOIs on the computer at the end of the week should be exactly those entered during the week, with expired TOIs showing as such.

Robustness

Metric The number of times the system crashes during a one week test period.

Tester As for reliability, the users will test this metric.

Data Screen prints of the error log for each system failure as well as a brief description, where possible, of the conditions that led to the crash.

Target There should be no system crashes.

Completeness

Metric The number of operations, functions or information items needed that are not catered for by the system.

Tester Again, this metric will be tested by the users.

Data A list of any information that was needed which was not on the system, as well as descriptions of any operations or functions required which the system did not provide.

Target The system should not be lacking in any of these areas.

User Friendliness

Metric This will be in the form of a report back from the users.

Tester The users.

Data A written report as to how easy the operator found the system to learn and use, whether he encountered any problems trying to do anything and any suggestions as to how he feels the system could be improved upon.

Target The user should be pleased with the system as it stands.

Response Time

Metric The system should take no longer than 5 seconds to display a TOI, and no longer than 10 seconds to load the viewer and display the instruction. Downloading of TOIs should not take longer than 15 minutes and access on the LAN should take no longer than 15 seconds (three times as long as from the local hard drive).

Tester The software development team can test this metric. The equipment used should be exactly the same as that which will be used at the Power Station. For the download and LAN access tests, the equipment at the power station should be used.

Data 65 tests on the local hard drive, 25 on the LAN and 10 downloads will be performed. These tests should be spread evenly throughout a 24 hour period during a regular working day, preferably a Monday or Tuesday when traffic on the LAN is usually at its heaviest. Response times should be monitored via a small computer program which measures the time between the user clicking either the 'View', 'Instruction' or 'Download' buttons, and the time it takes for the system to become stable once more.

Target No response times should be outside the thresholds set.

Maintainability

Metric The time it will take to identify and correct a problem should be estimated. (If there are no problems to be fixed within the program after the initial test week, some trivial 'test' maintenance routines, designed by someone from Duvha, could be performed.)

Tester An experienced software professional who is not a member of the software development team should do the time estimation while the actual maintenance should be performed by the software development team.

Data The actual time taken to identify and correct the problem.

Target The two values should be within 10% of each other.

Reusability

Metric The number of objects requiring modification for this program to be put into operation at another power station.

Tester The software development team should approach the Operations Manager at one or more of the other power stations to determine the needs of those power stations with regard to TOIs, perhaps demonstrating the test system at the same time. Based on these meeting it should be possible to estimate how far the program goes towards satisfying the needs of other power stations.

Data A list of all objects that would require modification for the program to work on other power stations.

Target This list should be as short as possible, with only objects that are power station specific appearing.

Efficiency

Metric The amount of disk space used and the CPU usage.

Tester The software development team and the computer centre staff at Duvha.

Data After the week of user testing it will be possible to get an indication of disk space requirements for the system. These can be extrapolated over any period to determine the optimum time before old TOIs should be archived to a historical database. CPU usage can be monitored using software.

Target The executable file should not occupy more than 1MB of disk space and should require less than xxxKB of RAM in order to execute. Data should not occupy more than xMB more of disk space per week.

Complexity

Metric This will take the form of a report written by a software professional who is not a member of the software development team.

Tester An employee of the organisation responsible for the development of the software.

Data A list of any algorithms, data flows or object classes which the tester feels are unnecessarily complex and suggestions as to how they can be improved. There should be no algorithms, data flows or objects which require modification.

Understandability

Metric The depth of understanding a completely independent software professional can gain as to how the system is implemented having had access to all the available documentation as well as to the test system for a week.

Tester One of the IT people based at Duvha Power Station was selected. This person is well-versed in Visual C++ and has a vague understanding of what a TOI is but has not been involved in the project in any way until the test period.

Data A report back as to how clear and understandable the system was to her as well as any questions she might have.

Target There should have been no problem in understanding the system and there should be no major problem areas.

Testability

Metric The number of time each operation for each object was accessed during the one-week testing period.

Tester The software development team should build routines into the software causing a record to be added to a database file containing information about the object identification and the current operation, every time an operation on an object is performed. The database files from all test sites, that is, the user sites as well as the developer sites, will be combined. It must be noted, however, that this could have an adverse affect on response time and system efficiency so the computers being used for those tests should preferably not have these routines activated.

Data A list of all system objects and their operations with the tally of the number of times that operation was accessed being obtained from the database file.

Target Each operation for each object should have been executed at least once.

Modifiability

Metric The time it will take to make one or more changes to the system should be estimated. (If there are no modifications to the program required after the initial test week, some trivial 'test' modifications, designed by someone from Duvha, could be performed.)

Tester An independent software professional, in this case, the same Duvha IT person who tested the system for understandability, did the estimate. The software team then made the modification.

Data The actual time taken for the modification to be performed.

Target The estimate time and the actual time should be within 10% of each other.

Interoperability

Metric how well the system conforms to the Project R guidelines at Duvha.

Tester Someone well-versed in Project R, possibly in conjunction with the software developers.

Data A list of any guidelines to which the system either does not conform, or to which it conforms incompletely.

Target There should be no areas of non-conformance.

A detailed code inspection will also be carried out at this point. The code will be checked for comments, indentation, variable and function name usage and module length. In this case, the following standards apply:

Comments must appear at the declaration of each class, describing the class and its operations. Comments should also appear with the code for each operation explaining what is being done. Any other areas where the meaning of the code or the reason behind it may be obscure should also be commented upon.

Indentation All blocks should be indented by a tab.

Variable Names These should be descriptive, i.e. it should be possible to tell from the variable name what the variable represents. The variable type should be represented by the first letter of the name, for example, sOriginator_Name would be the name of the Originator of the TOI and would be of string type.

Module Length A module should not exceed a page in length.

QAT53: IMPLEMENTATION - Development

The results of QAT52 should be thoroughly analysed. In this case a meeting will be held between the software development Project Manager, the software development Project Leader, one of the programmers, the Duvha IT professional who was involved in the testing, one or more of the operators who were involved in the testing and the Operations Manager of Duvha to discuss the overall performance of the system. If problem areas are identified, an earlier cycle will be reverted to. In most cases, problems identified at this late stage are not serious and will not require a large amount of rework. If there are very serious problems resulting in many modifications, it is clear that the software quality control process during earlier cycles was not operating correctly and why this is the case should also be addressed in detail. The problem areas will then be retested in the same manner and by the same people as before. If they are then considered to be of an acceptable quality, the system can be commissioned.

QAT54: IMPLEMENTATION - Review/Planning

In addition to the revised SQAP and the Object Model, Dynamic Model and Functional Model review documents. A document containing the results of all tests and an evaluation of the quality of the system should now be drawn up. The following is just an example and is not detailed.

QUALITY ASSURANCE REQUIREMENTS PLAN FOR DUVHA POWER STATION TEMPORARY OPERATING INSTRUCTION (TOI) PROGRAM

Test results

Security

This criterion was tested by an employee of the company which developed the software, who is well versed in Windows, DOS, Novell Netware, Unix and has a fair understanding of Visual C++. He was unable to add or tamper with the TOIs at all.

Reliability

The system operated successfully at Duvha Power Station for the period of one week. Although the LAN went down on numerous occasions, this did not affect the TOI system at all. The TOIs on the system at the end of the week were identical in all respects to the paper copies.

Robustness

The system crashed on 2 occasions.

1. The system crashed due to memory problems as the operator had accidently loaded 4 DOS windows while trying to do something else. This problem should not reoccur as operators become more familiar with the workings of Windows.
2. The system crashed when an operator tried to download TOIs from the LAN and the LAN was down. The program now checks the availability of the LAN before attempting to download TOIs.

Completeness

The Senior Shift Supervisor asked whether the TOI date could be automatically loaded with the system date when a TOI is added to the system as in most cases this would be the correct date. This has been done.

User Friendliness

The operators did not like having to double click on the minus sign in the top left corner in order to exit from screens so an 'OK' button has been added to the system to enable them to exit by clicking on this.

Response Time

All of the tests were within the limits set.

Maintainability

It was estimated that to get the system date to be automatically entered into the TOI date field would take one hour. The actual time taken, including recompiling and testing was actually 1 hour and 5 minutes which was just within the 10% leeway.

Reusability

The Operating Managers of Kendel, Kriel and Matla power stations were all very impressed with the TOI program and felt that it could be adopted as it stands for their power stations.

Efficiency

The executable file occupies 511Kb of disk space which is negligible. After 1 week of testing, the data occupied 712Kb of disk space. Thus it is estimated that if old TOIs are archived on an annual basis there should be no disk space problem. The CPU usage was also within acceptable limits.

Complexity

In the opinion of an IT consultant at Megawatt Park, the system is in no respect unnecessarily complex.

Understandability

The Project Leader at Duvha Power Station understood the complexities and manner of operation of the system in all respects after studying it and all the related documentation for 1 week, even though she is not familiar with Visual C++, only with Borland C for DOS.

Testability

According to the statistics gathered from three of the test sites, all operations on all object were executed at least 4 times.

Modifiability

It was estimated that to add an OK button to two of the screens, including recompilation and testing, would take approximately 30 minutes. In actuality it took 27 minutes.

Interoperability

The system conforms in all respects to the Project R guidelines.

Code Inspection

The source code was inspected by programmers not involved in this project. A few additional comments were requested, but otherwise, all source code was deemed of acceptable quality.

EVALUATION

The TOI program was accepted by Duvha Power Station after 1 pass over the Spiral model.

6.6 Summary and Conclusions

Due to the limited scope of this project, the demonstration of SQARMOO has been necessarily very superficial. The project used for demonstration purposes is small and no account has been made of many of the quality assurance techniques mentioned in Chapter 2, such as CASE tools or automatic verification techniques. Metric 'values' have been fairly arbitrarily described with no mathematical metrics being used. This does not mean that SQARMOO does not work under these conditions, but rather that they were not dealt with here. It could perhaps be taken up as an Honours level project to apply SQARMOO more rigorously to a more substantial project.

It should also be noted that in the scenario described in this chapter, certain meanings have been applied to quality criteria terminology. These meanings are by no means the only acceptable ones for those criteria. Depending on the organization, an entirely different meaning may be assigned to a criterion which has the same name. For example, we have used the term 'Reusability' to indicate whether or not the system will be easy to implement at other power stations. Reusability can also mean whether or not certain system classes may be reused in other, completely diverse projects, or how many classes used in the current project originated from outside projects, or even how often a module is reused within the same software program. How the term is interpreted depends entirely on how the user and the software developer want it to be determined. This also applies to how it is measured. These aspects of software quality can not and should not be prescribed. However, this does place an additional burden on the Quality Assurance Team as it is, therefore, up to them to ensure the exact and correct meaning of each criterion is understood by the user and the software development team.

Various factors influence the quality of a delivered software product. These factors include the time that was allowed for the development of the product, the budget available, the importance of the product and the consequences of system failure. There is also a major difference in the perception of quality by different people although everyone agrees that quality does in some respect have a bearing on conformance to requirements. Many organizations perceive quality as an intuitive part of the development of software and make no special efforts towards achieving it. This is not a good idea. Unless everybody concerned with the project knows what goals they are working towards as regards qua-lity, quality will never be satisfactorily achieved. Thus, it is necessary to have some form of reference model on which to base these quality goals.

Because object-oriented software development emphasises the object rather than the procedure, as in conventional software development, it is believed that following an object-oriented methodology, the OMT methodology proposed by Rumbaugh et al being the one used for this project, will result in software that is more true-to-life, more modular and which will be easier to reuse. Object-oriented development is an iterative process and thus the Revised Spiral Model of Software development was selected as the life cycle model to be used.

Existing Quality Assurance Methodologies were examined and consolidated to form the basis for a new reference model dubbed the Spiral Quality Assurance Reference Model for Object-Orientation (SQARMOO). This model is based on the Revised Spiral Life Cycle Model proposed by du Plessis and van der Walt (1994) and the Object Modelling Technique proposed by Rumbaugh et al (1991). A Quality Assurance Matrix is proposed consisting of 20 Quality Assurance Tasks, one for each quadrant in each cycle of the Spiral. A brief description of the tasks is given. This model could easily be expanded upon to include some of the other aspects of quality assurance described in Chapter 2 of this dissertation, such as the use of CASE tools and the Total Quality Management philosophy.

A software development scenario was described with the various quality assurance tasks proposed under the SQARMOO model being successfully demonstrated.

It is, therefore, concluded that :

- quality in software is achievable by using a methodological approach, sound engineering principles and rigorous project management techniques.
- the analysis and design tools and methods as well as the programming language used can influence the quality of a software project.
- the object-oriented methodology assures quality in software to a far greater extent than other methodologies.
- testing plays a very important role in assuring quality.
- the management of a software development project goes a long way to affecting the quality of the result.

REFERENCES

BOEHM, B.W.; **"A Spiral Model of Software Development and Enhancement"**, IEEE COMPUTER, May 1988, p61-72

BOOCH, G.; **"Object-Oriented Design - with applications"**, Benjamin-Cummings, USA, 1991

CHIDAMBER, SHYAM R.; KEMERER, C. F.; **"Towards a metrics suite for Object-Oriented Design"**, Communications of the ACM, 1991, p 197-211

COAD P., Y. E., **"Object-Oriented Analysis"**, Prentice-Hall, 1990

DELEN, G.P.A.J. and RIJSENBRIJ, D.B.B.; **"The Specification, Engineering and Measurement of Information Systems Quality"**, Journal of Systems and Software, 1992, p205-217

DEMING,W.E.; **"Out of the crisis"**, MIT Centre for Advanced Engineering Study, Cambridge, Mass 1986

DENNING, P. J.; **"EDITORIAL: What is software quality"**, Communications of the ACM, Vol. 35 No 1, January 1992, p13-15

DUNN, R. H.; **"Software Quality: Concepts and Plans"**, Prentice Hall, 1990

DU PLESSIS, A.L.; **"CAISE: The Opportunity and the Challenge"**, Inaugural lecture on assuming the position of professor in the Department of Computer Science and Information Systems at the University of South Africa, August 1992

DU PLESSIS, A. L. & VAN DER WALT, E; **"Modelling the Software Development Process"**, IFIP WG8.1 Working Conference on "Information Systems Concepts: Improving the Understanding", April 1992

DU PLESSIS, A. L. & VAN DER WALT, E;
"A Revised Spiral Model for Object-Oriented Development",
submitted to CAISE'94 6th Conference on Advanced Information Systems Engineering, The Netherlands, June 1994

ESKOM; **"Quality Requirements for Related Quality Services"**,
EVS010, Rev 0, Jan 1989, 16pp

FENTON, N. E.; **"Software Metrics: A Rigorous Approach"**,
Chapman and Hall, 1991

GILLIES, A.N. C., **Software Quality**,
Chapman and Hall, 1992

GLASS, R. L.; **"Building Quality Software"**,
Prentice Hall, 1992

GILB, T.; **"Principles of Software Engineering Management"**,
Addison-Wesley, 1998

GRAHAM, D. R.; **"Testing and quality assurance - the future"**,
Information and Software Technology, Vol. 34 No. 10, Oct 1992, p 694-697

HALL, P. A.V., **"Software Development Standards"**,
Software Engineering Journal, May 1989, p 143 - 147

HENDERSON-SELLERS, B;
"Parallels between object-oriented software development and total quality management",
Journal of Information Technology, vol.6 no. 2, 1991, p63-67

ISHIKAWA,K
"What is Total Quality control? - the Japanese Way (translated by D.J.Lu)"
Engelwood Cliffs, NJ, Prentice Hall, 1985

KEENE, S. J.; **"Cost Effective Software Quality"**,
Proceedings of the Annual Reliability and Maintainability Symposium, 1991, p433-437

KITCHENHAM, B. A.; WALKER, J. G.;
"A Quantitative Approach to Monitoring Software Development",
Software Engineering Journal, January 1989, p2-13

LARANJEIRA,L.A.;
"Software Size Estimation of Object-oriented Systems"
IEEE Transactions on Software Engineering, 1990, p510-521

MACRO,A & BUXTON,J.,
"The Craft of Software Engineering",
Addison-Wesley, 1987

MOREAU, D.R & DOMINCK W.D;
"Object-/oriented Graphical Information System: Research Plan & Evaluation Metrics",
Journal of System & software vol 10, 1989, p23-28

MEYER, B.; **"Object-Oriented Software Construction"**,
Prentice-Hall, 1988

MORETTI, R; **"SDL and Object-Oriented Design: A Way of Producing Quality Software"**,
CSELT Technical Reports, vol. 18 no. 2, 1990, p 131-134

PARSAYE, CHIGNALL, KHOSHAFIAN & WONG,
Intelligent Databases,
Wiley, 1990

REDIG, G and SWANSON, M.,

"Total Quality Management for Software Development",
Proceedings of IEEE 1991 National Aerospace and Electronics Conference; Vol 3, 1991, p1282-1288

RUMBAUGH J, BLAHA M., PREMERLANI W., EDDY F., LORENSEB W.;
Object-Oriented Modelling and Design,
Prentice-Hall, 1991

SAGE, ANDREW P. & PALMER, JAMES D. ;
Software Systems Engineering,
Wiley, 1990

SCHULMEYER, G.G. & McMANUS, J.I.;
Handbook of Software Quality Assurance 2nd Edition,
Van Nostrand Reinhold,1987

SCHACH, S.R.; **Software Engineering - Second Edition,**
R.D. Irwin Inc. and Aksen Associates, Inc, 1993

SMITH. M.D. AND ROBSON,D.J.;
"A framework for testing object-oriented programs",
Journal of Object-oriented Programming, Vol.5 No 3, 1992 p45-53

South African Beau of Standards ISO 9000-1
South African Beau of Standards ISO 9000-3

STAKNIS, M.E.;
"Software quality assurance through prototyping and automated testing"
Information and Software Technology, Vol 32 No 1, 1990 p26-33

NEUMANN,E;
"Risks to the Public"
ACM Sigsoft, Software Engineering Notes, Vol 19 No 1, Jan 1994

TEGARDEN, D. P., SHEETZ, S. D. & MONARCHI D. E.;
"Effectiveness of Traditional Software Metrics for Object-Oriented Systems", Proceedings of the 25 th hawaii confrence on System Sciences vol 4, 1992, p359-368

THAYER, R.H., PYSTER, A, and WOOD R.C.;
"The challenge of Software Engineering Project Management", IEEE Computer, Vol 13 No 8, Aug 1980, p 51-59

TRAMMELL, C. J. & POORE, J. H.;
"A Group Process for Defining Local Software Quality: Field Applications and Validation Experiments",
SOFTWARE - Practice and Experience, Vol. 22 No. 8, 1992, p 603-636

WALLACE, D. R. & FUJII, R. U.;
"Software Verification and Validation: An Overview",
IEEE Software, May 1989, p 10-17