A Quality Assurance Reference Model for Object-Orientation

DEBORAH M. THORNTON AND ANNETTE L. STEENKAMP
Department of Computer Science and Information Systems
University of South Africa. PO Box 392. Pretoria

Abstract: Software quality assurance for object-oriented information systems development is the issue dealt with here. A Quality Assurance Reference Model is proposed based on the Revised Spiral Life Cycle Model. The Quality Assurance Reference Model associates quality criteria, quality factors and metrics into a matrix framework that may be used to achieve quality assurance for all cycles of the Revised Spiral Model.

Keywords: Software Quality Assurance, Object-Orientation, Spiral Life Cycle Model, Quality Criteria, Quality Factors, Metrics, Quality Assurance Tasks

Computing Review Category: Research.

1. Introduction

A quarter of a century after the identification of the 'Software Crisis', software is still delivered behind schedule, over budget and with more errors than functionality [10]. There are a number of reasons for this:

- **No measurable targets are set.** It is very difficult to set targets for the assessment of the quality of a software system. Should software be evaluated 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 problems are caused by the system once it is in operation. As Glib points out [12]: "Projects without clear goals will not achieve their goals clearly."
- **The increasing complexity of contemporary software.** 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 complicated functions, very often in real-time and in a variety of distributed locations.
- **Rapidly changing hardware and software.** Keeping up to date with fast-changing technology is difficult and time-consuming.
- **Software quality is difficult to assess.** People tend to define software quality differently.

In the following sections, we investigate the issues contributing to the achievement of quality in software development. Section 2 addresses important considerations regarding the quality conundrum from an object-oriented perspective, while section 3 proposes a quality assurance reference model.

2. Achieving Software Quality

In order to achieve quality in software development, the whole development process must be addressed. A methodology and a framework for the the development process from requirements analysis right through to maintenance should be identified. Managerial as well as technical tasks must be specified.

2.1 The Object-Oriented Methodology for Software Development

Software development 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 development is a methodology. A methodology consists of a set of methods which guide the processes of creating the solution. One type of methodology is based on the object-oriented paradigm.

As about two-thirds of total software costs are devoted to maintenance, software should 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. The object-orientation characteristics of data encapsulation, inheritance and polymorphism go a long way to ensuring that such modularity in a system is achieved. A system designed according to the principles of object-orientation should thus be far easier, and therefore cheaper, to modify and maintain.

It is commonly known 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, the modelling of enterprises and the production of understandable documentation. An object-oriented development methodology encourages software developers to work and think in terms of the application domain through most of the software development life cycle (SDLC).

The efficiency of object-oriented programming languages is still in question [6], 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 [20] this should not be viewed as a major issue. 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. An OO methodology has therefore been identified as the preferred methodology for the purposes of this investigation.

2.2 The Assessment of Software Quality

Another consideration with respect to software quality is the ability to measure quality during the development process as well as that of the finished product. This requires the drafting of a framework within which aspects of quality can be measured. This framework consists of criteria which the software must satisfy, and metrics to measure the degree to which the criteria are satisfied. 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 [10].

2.2.1 Software Quality Criteria

In order to assess the quality of software, it is necessary to first clearly define what it is we are looking for. A good way of doing this is to define quality criteria with targets which the software must measure up to. These criteria 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 being a synthesis based on ideas from numerous sources including [6], [10] & [12]. These criteria may be divided into two broad categories: technical quality assessment criteria (table 1) and user quality assessment criteria (table 2).

a. Complexity	The Complexity of the algorithms used as well as of the target system.
b. Understandability	Ease of understandability of the system developer's intentions.
c. Maintainability	Ease with which the system can be corrected, adapted and/or enhanced over its lifetime
d. Reusability	Ability to make use of code or objects in other systems.
e. Efficiency	How well the system makes use of the available resources.
	Ease of testing as much as possible of the functionality of the system in real-life conditions.
f. Portability	Ability to move a system across operating environments.
g. Modifiability	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.
i. Interoperability	Whether the system can communicate with other software packages e.g. import or export data or launch other programs.

Table 1: Technical Quality Assessment Criteria

Once the most relevant quality criteria have been identified for a particular system, importance weights for each criterion must be decided upon. Thereafter, operational methods for determining the scores for the criteria within specific software development approaches must be determined.

2.2.2 Software Metrics

The purpose of software quality metrics is to make assessments throughout the SDLC as to whether the software quality requirements are being met [22]. 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.

a. User-friendliness	Ability of the system to be easily understood and used by human users.
b. Response Time	Response time of the system must be acceptable
c. Reliability	Degree to which the system can be relied on to be available when needed and to product correct information
d. Robustness	Degree to which the system is able to withstand incorrect usage or conditions without failing.
e. Correctness	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	Extent to which a product meets its constraints
h. Security	Avoidance of unauthorised access.
i. Flexibility	Ability of the system to satisfy different user requirements
j. Completeness	The system should satisfy all the user requirements, not just a subset of them.
k. Utility	Extent to which a user's needs are met when a correct product is used under conditions permitted by its specifications.

Table 2: User Quality Assessment Criteria

2.3 Quality within the Software Development Process

A SDLC model structures the development process into a number of phases. There are a number of different models currently favoured by developers. Probably the most well-known is the *Waterfall Model.* Other models are the *Code-and-Fix Model*, the *Iterative* or *Evolutionary Model, the Transform Model* and the *Rapid Prototyping Model.*[10]. A *Spiral Model,* which incorporates many of the strengths of the other models while resolving many of their difficulties has also been proposed [1]. The underlying concept of this model is that each cycle in the development process involves a cyclic progression involving the same sequence of steps for each part of a product and for each of its levels of elaboration.

The major advantages of the spiral model are :

- Early focus on options involving the reuse of existing software.
- It promotes preparation for later modification to the system.
- Because all types of objectives and constraints are identified during each round of the spiral, software quality control is facilitated.
- It focuses on eliminating errors and unattractive alternatives early.
- 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.
- A viable framework for integrated hardware and software system development is provided.

The Spiral Model has been revised to complement object-oriented development [7] & [8]. This model proposes a three-dimensional view of the software development process.

Firstly, software development is viewed at three levels of abstraction:

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

The Model can also be viewed in terms of four quadrants of a cycle:

- Issue Formulation during which the objectives, needs, constraints and alternatives are considered with respect to satisfying the stated objectives for a cycle.
- Risk Analysis and Evaluation of Alternatives during which the proposed strategies are evaluated and the risks involved are identified.
- 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.
- 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.

Finally, there are five cycles in the Spiral Model:

- Feasibility Cycle: This cycle starts with the formulation of a problem statement and culminates with a Project Proposal.
- Architecture Cycle: In this cycle the top level system software and hardware architectures are determined.
- Analysis Cycle: The object-oriented analysis, using the diagrams defined during the previous cycle, is performed here.
- Design: the system is designed and sub-systems are developed during this cycle.
- Implementation: The system design is implemented and tested.

3. SQARMOO : A Spiral Quality Assurance Reference Model for Object-Orientation

The proposed quality assurance reference model for object orientation closely follows the pattern of the Revised Spiral Model for object-oriented development and is therefore referred to as the Spiral Quality Assurance Reference Model for Object-Orientation or SQARMOO.

In terms of this model, quality assurance can also be viewed at three levels, namely Universal, Worldly and Atomic for project management of object-oriented software development [8] (Table 3). The Universal Level provides a global view of the whole quality assurance process taken by senior management. Senior management will. firstly, decide whether or not to apply SQARMOO (or any other reference model) at all to the project. They will then have to decide to what extent they wish to go in applying a quality assurance reference model. For example, what support tools they are prepared to consider, whether new hardware and software would be feasible, whether it would be practical to set up a separate Software Quality Assurance team, and other such related matters. Middle management is concerned with a more detailed approach to quality assurance at the Worldly level, for example, which quality criteria apply to the current project and what the target values for those criteria should be, while junior management is concerned with the minutiae of achieving, measuring and controlling quality at the Atomic level. It is the task of Junior management to decide on how the quality criteria decided upon by Middle management will be achieved as well as how to measure the extent to which they have been achieved and to see to it that these measures are made accurately and correctly.

LEVEL	DOMAIN	QA TASKS
Univeral	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 3: Provisions of a Methodology for QA

The Revised Spiral Life Cycle Model has software development as an iterative process, taking place over five cycles. In terms of quality assurance, SQARMOO has the quality process also taking place in an iterative manner. Although each cycle and quadrant has specific quality assurance tasks assigned to it, at any time, if it appears that the quality of the system is being compromised because of incorrect or irrelevant quality assurance techniques, it is possible to loop back to a previous cycle and rectify the problem. During Cycle 1, the feasibility of the project as a whole and of the ability to enforce quality standards in particular, is reviewed, usually by senior and middle management. It is during this cycle that it must be decided whether a quality assurance reference model is appropriate for the project and which support tools and techniques are to be used to measure quality during the development process. Cycle 2 is concerned with the architecture of the proposed system and its effect on the overall quality of the system. It must be assessed whether the software and hardware decided on pose any problems from a quality assurance perspective, and if they do, steps must be taken at this point to rectify these problems. During Cycle 3, the Analysis Cycle, the quality criteria which are applicable to the project and their target values are identified. Cycle 4 deals with the design of quality assurance techniques to ensure the achievement of the proposed target values for the criteria, and the identification of relevant metrics to measure whether or not this has, in fact, been the case. Finally, during Cycle 5 the techniques and methods identified in the previous four cycles are applied.

Quality assurance issues are addressed during each of the four quadrants. The first Quadrant deals with the formulation of standards in terms of the current cycle. During Quadrant 2, metrics, testing tools and other quality assurance techniques are analysed for use in measuring the achievements of the cycle. 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.

3.1 Software Quality Levels

Each of the software quality levels is explained further in this section.

3.1.1 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 global quality criteria that are important for the project are established by senior management. Priorities and weighting factors are also allocated to the various criteria at the Universal level. Associated with each criterion, if possible, is a target value as shown in table 4. For example, management may consider the quality criterion of the

maintainability of the system to be 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. It may, however, not always be possible to associate target values to criteria at this level.

Quality Criteria	Criterion 1 Cn	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

$Q = \{ C_1TV_1, C_2TV_2, ..., C_nTV_n \}$

where Q represents the quality of the system which can be determined by the association of various criteria (C_n), and their target values (T_n).

Table 4: Quality Criteria at the Universal Level

3.1.2 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 although this may not be possible in all cases (table 5).

To expand on the Maintainablity example, if the target value set at the Universal Level was, in fact that, 'no corrective action should take more than 48 hours to implement', it must now be ascertained how this applies to the quality factors of 'correctability', 'testability' and 'expandability'. In this case, no new sub-target values need be set as the 48 hour target value is still relevant for these factors collectively.

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

Quality Criteria	Criterion 1 C_1			
Factors	Factor 1 F_1	Factor 2 F_2	...	Factor n F_n
Target Values	Target Value TV_1			
Sub-Values	Value 1 STV_1	Value 2 STV_2	...	Value n STV_n

$C_1 = \{ F_1, F_2, ..., F_n \}$

Each quality criterion, C_n, may be decomposed into various quality factors, F_1, F_2, ... F_n.

$TV_1 = \{STV_1, STV_2, ..., STV_n \}$

Each Target Value, TV_n, may be decomposed into various sub-target vlaues, STV_1, STV_2,...STV_n.

Table 5: Quality Factors at the Worldly Level

3.1.3 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 (table 6). It must be noted that 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.

Quality Factors	Factor 1 F_1			
Metrics	Metic 1 M_1	Metric 2 M_2	...	Metric n M_n
Sub_Values	Value 1 STV_1			
Metric Values	Metric Value 1 MV_1	Metric Value 2 MV_2	...	Metric Value 3 MV_3

$F1 = \{ M_1, M_2, ..., M_n \}$

$STV1 = \{ MV_1, MV_2, ..., MV_n \}$

Table 6: Quality Metrics at the Atomic Level.

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

A matrix of Quality Assurance Tasks (Table 7) can be mapped against the Revised Spiral Life Cycle Model (Figure 1), with each task being associated with a particular quadrant within a particular cycle.

CYCLES	QUADRANTS 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

Table 7: Measures of Software Quality

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 user 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 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, 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 in terms of quality 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.

QAT43: DESIGN - Development.

The system should also be evaluated fully with respect to all the quality criteria mentioned in the SQAP. Where possible, these should be decomposed into factors and target values for these factors (called sub-target values to distinguish them from the target values for the quality criteria decided on in the previous cycles) should be assigned where possible.

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 quality factor defined in the SQAP, determine the metrics that must be measured, how this is to be achieved and any assumptions made. 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 collection of data and measurement of quality metrics 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.

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 programmer'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.

4. Summary and Conclusions

A Spiral Quality Assurance Reference Model for Object-Orientation (SQARMOO) is developed. Quadrants within cycles are mapped to a matrix with each element detailing a separate Quality Assurance Task (QAT). Quality is assessed by defining criteria, with both user and technical considerations in mind, which the software must meet. The criteria can usually be decomposed into quality factors which can be further decomposed into quality metrics which are measurable properties of one or more of the quality criteria that we are seeking to measure.

Measuring and ensuring the quality of a software development project can be an expensive and involved task. How expensive and involved it is should be determined by the possible consequences if the system were to fail. The most important consideration is that the necessity for quality assurance must first of all be recognised by an organisation and the commitment to ensure quality be adopted by all members of that organisation. How the quality assurance reference model is applied is the concern of that organisation alone.

REFERENCES

1. Boehm, B.W. 1988. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, May 1988: 61-72.
2. Booch, G. 1991. *Object-Oriented Design - with applications*. USA: Benjamin-Cummings.
3. Chidamber, S.R. & Kemerer, C.F. 1991. Towards a metrics suite for Object-Oriented Design, *Communications of the ACM*, p 197-211.
4. Coad, P. & Yourdon, E. 1990. *Object-Oriented Analysis*. Prentice-Hall.
5. Denning, P.J. 1992. EDITORIAL: What is software quality? *Communications of the ACM*, 35(1):13-15.
6. Dunn, R.H. 1990. *Software Quality: Concepts and Plans*. Prentice Hall.
7. Du Plessis, A.L. & Van Der Walt, E. 1992. Modeling the Software Development Process. *IFIP WG8.1 Working Conference on "Information Systems Concepts: Improving the Understanding*
8. Du Plessis, A.L. & Van Der Walt, E. 1994. A Revised Spiral Model for Object-Oriented Development, submitted to *CAISE*94 6th Conference on Advanced Information Systems Engineering*, The Netherlands, June 1994
9. Fenton, N.E. 1991. *Software Metrics: A Rigorous Approach*. Chapman and Hall.
10. Gillies, A.C. 1992. *Software Quality*. Chapman and Hall.
11. Glass, R.L. 1992. *Building Quality Software*. Prentice Hall.
12. Glib, T. 1989. *Principles of Software Engineering Management*. Addison-Wesley.
13. Henderson-Sellers, B. 1991, Parallels between object-oriented software development and total quality management. *Journal of Information Technology*, 6 (2):63-67
14. Keene, S.J. 1991. Cost Effective Software Quality. *Proceedings of the Annual Reliability and Maintainability Symposium*. p433-437
15. Kitchenham, B. 1989, Software Quality Assurance. *Microprocessors and Microcomputers*, 13(6):373-381
16. Laranjeira, L. 1990. Software Size Estimation of Object-Oriented Systems. *IEEE Transactions on Software Engineering*, p 510-521
17. Macro, A & Buxton, J. 1987. *The Craft of Software Engineering*. Addison-Wesley.
18. Meyer, B. 1988. *Object-Oriented Software Constructon*. Prentice-Hall.
19. Moretti, R. 1990. SDL and Object-Oriented Design: A Way of Producing Quality Software. *CSELT Technical Reports*, 18(2):131-134
20. Runbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorenseb,W. 1991. *Object-Oriented Modelling and Design*. Prentice-Hall.
21. Sage, A.P. & Palmer, J.D. 1990. *Software Systems Engineering*. Wiley.
22. Schulmeyer, G.G. & McManus, J.I. 1987. *Handbook of Software Quality Assurance 2nd Edition*. Van Nostrand Reinhold.
23. Schach, S.R. 1993. *Software Engineering - Second Edition*. R.D. Irwin Inc. and Aksen Associates, Inc.
24. Staknis, M.E. Software quality assurance through prototyping and automated testing. *Information and Software Technology*, 32(1):26-33
25. Tegarden, D.P., Sheetz, S.D. & Monarchi, D.E. 1992. Effectiveness of Traditional Software Metrics for Object-Oriented Systems. *IEEE*, p359-368
26. Trammell, C.J. & Poore, J.H. 1992. A Group Process for Defining Local Software Quality: Field Applications and Validation Experiments. *SOFTWARE - Practice and Experience*, 22(8):603-636