A framework for the teaching of software quality assurance at the undergraduate level
Y.-W. Chan & H.-C. Ho

Department of Computer Science, City Polytechnic of Hong Kong, Kowloon, Hong Kong

ABSTRACT

As the discipline of software engineering matures, there is an increasing emphasis on software quality assurance. Naturally, this trend has to be reflected in the way software engineering is taught, especially in tertiary institutions. The "Guidelines for the application of ISO 9001 to the development, supply and maintenance of Software", released by the international Standard Organisation in 1991, provides a timely source of reference on this important issue for computer science educators. This paper examines the significance of the ISO standards with regard to software. Based on the ISO 9000 recommendations, a framework for teaching software quality assurance concepts and techniques is presented. The incorporation of the framework into an existing undergraduate computing programme at the City Polytechnic of Hong Kong is then discussed. The paper concludes with some observations regarding the future enhancements of the curriculum.

I. INTRODUCTION

Due to the ever expanding need of information processing and repaid advance in information technology, the problems of poor quality software have become increasingly complex and critical. Costs of poor quality software, just to name a few, include low productivity, unnecessary maintenance expense, lost opportunity, and unrealized savings.

How can we develop high-quality software? Software engineering emerged as the solution emphasising the application of engineering principles to software. Most current computer science curricula include some coverage of software engineering topics: project management, requirement analysis, system design and implementation, and software quality assurance. Unlike other software engineering topics which specify procedure or discipline for a particular phase of development, software quality assurance involves planning, measuring and monitoring software development activities throughout the software development life cycle.
II. QUALITY STANDARD

The quest for quality in every aspect of our life can best be summarised in statements made by George Bush, the ex-president of USA -

"Quality management is not just a strategy. It must be a new style of working, even a new style of thinking. A dedication to quality and excellence is more than good business. It is a way of life, giving something back to society, offering your best to others".

A series of standards, both for award and certification, have been developed by various organizations to tackle the widespread problems relating to quality. The ISO 9000 series of quality standards published by the International Standard Organization, has tremendous impact in many industrial and economic sectors. The following ISO 9000 series and related standards listed in chronological order provide a summary for reference:

ISO 8402 Quality Vocabulary. (1986)
ISO 9000-3 Quality Systems and Quality Assurance Standards - Guidelines for the application of ISO 9001 to the development, supply and maintenance of software. (1991)

In addition, an ISO Technical Committee called Ad Hoc Task Force of TC176, which was commissioned to prepare a strategic plan for ISO 9000 series standard has also reserved ISO 10000 series standard for technical aspects such as audit requirements of a quality system. The task force report which has been known as "Vision 2000" for its forward planning vision in the quality arena by the year 2000, shows us how the evolving system of standards meets the combined needs for quality management and quality assurance from both the producer's and the purchaser's viewpoints.

The ISO 9000 series standards has been promptly adopted by many nations and regional bodies for guidance, certification, auditing, and documentation of quality systems. As quality assurance continues to be a competitive weapon for companies, software quality assurance inevitably will become more important to software participants.
III. SOFTWARE QUALITY ASSURANCE (SQA)

Software has been classified in the Vision 2000 report as one of the four generic product categories which include hardware, software, processed materials, and service. It should be noted that software development involves project management which combine many characteristics of a service together with production and/or delivery of hardware and/or software. Software company should then be classified in the industry sectors which offer all four generic product except processed materials.

However, the process of development and maintenance of software is different from that of most other types of industrial products. The nature of software development is such that some activities are related to particular phases of the development process, while others may apply throughout the process. The ISO 9000-3 software process model is shown in the following diagram:

Based on this framework of software quality assurance, we can see that the study of SQA must include technical knowledge as well as management skills. The quality management approach adopted by ISO 9000 which emphasizes planning and process management also provide a more comprehensive picture on the factors affecting the success of software process.
IV. A FRAMEWORK FOR SOFTWARE QUALITY EDUCATION

The importance of software quality underscores the need for practitioners in the software industry to address SQA issues. Recent developments in the certification and registration of quality systems by accreditation organizations and governments represent concerted efforts in this direction. At the same time, it is important to get the message across to tomorrow's professionals. It is here that computer science educators have an important role to play. SQA concepts, principles and techniques have to be incorporated into the curriculum, so that graduates are well prepared for the challenges ahead of them. Our framework for teaching SQA is based on the ISO 9000 quality standards, and puts emphasis on four aspects, namely, software quality, software process, configuration management, and measurement of software quality throughout the software life cycle. A summary discussion of these aspects is presented below.

1. Software Quality

The first question about software quality is what do we mean by quality? Quality is defined as "The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs." in ISO 8402-1986: Quality Vocabulary. Using this definition, we can define software quality as the totality of features and characteristics of a software that bear on its ability to satisfy stated or implied needs. However, diverse quality needs among different users of a software product together with invisibility, complexity, and easy-to-be-changed nature has caused many misconception on software quality. Examples of these include: software quality is intangible and not measurable; quality product are too expensive; quality problems are originated by programmer.

Causes of these misconceptions are very complicated. However, it is clear that a subjective view of software quality, which equates software quality to quality of artistic product like music or painting would not solve the problem. In addition, it is a requirement in the ISO 9000-3 standard that a quality assurance plan should be prepared in the project planning phase of project development. The quality assurance plan includes, among other items, quality objectives expressed in measurable terms whenever possible. Hence, the hierarchical quality model [3] is adopted to provide a model to quantitatively measure software quality. In the model, McCall and Cavano have proposed a useful categorization of factors that affect software quality. Those software quality factors focus on three important aspects of a software product: i) its operational characteristics, ii) its ability to undergo change, iii) its adaptability to new environment. Furthermore, a number of criteria which, represent software-oriented quality attribute and can be quantitatively measured using some software quality metrics, will be used to evaluate these factors. For example, targeting reusability as a quality objective, a hierarchical model of reusability is as follow:
Managing Quality Systems 605

factor (management-oriented view of software quality)

reusability which is defined as the extent to which a program can be used in other application.

criteria (software-oriented attributes which provide quality)

generality, modularity, machine independence

metrics (quantitative measures of those attributes)

stability measure, complexity measure, check list

Difficulties remain as there are currently no universally accepted measure of software quality. It has generally been agreed that in order to be able to assure, control or improve quality, it is necessary to be able to evaluate it.

2. Software Process

The hierarchical model of software quality provides a base for product control. However, the quality of any product is governed by the quality of the process used to develop it. To allow continuous process improvement, one must possess knowledge of the process as well as product. Thus for SQA, software process is the set of activities, methods, practices, and transformations that integrate managers and software engineers in using technology to develop and maintain software. To evaluate the software process, Capability Maturity Model (CMM) [2] can be used to provide a conceptual framework to help organizations to characterize the maturity of their process as well as to establish goals for process improvement.

In order to be able to assure process quality, ISO 9000-3 adopts a generic software process model in which software activities are divided into life-cycle activities and supporting activities. [12] The life-cycle activities include:

1) contract review
2) purchasers' requirements specification
3) development planning
4) quality planning
5) design and implementation
6) testing and validation
7) acceptance
8) maintenance
Supporting activities which are not phase-dependent are

1) configuration management
2) document control
3) quality records
4) measurement
5) rules, practices and conventions
6) tools and techniques
7) purchasing
8) included software product
9) training

In the context of SQA, it is necessary to be able to identify the input and output of all these software development activities. Thus product quality can be controlled. In the model shown in the previous diagram, examples of software products include: Tender, Contract, Request for Proposal, Functional Specification, Development Plan, Quality Assurance Plan, Configuration Management Plan, Design Documentation, Program, Test Plan, User Documentation, Acceptance Test Plan, and Maintenance Plan.

3. Configuration Management

Software development invariably entails constant changes to the software products which constitute the software system. It is necessary to identify, control, and track the versions of each software product. Configuration management is the management of those changes. We define a configuration item as a software product which has been approved and accepted as a deliverable. A baseline is a set of configuration items which together show that a project milestone is reached. Thus configuration management is the operation to accept into the baseline those items which have passed quality control. Typical baselines as it was shown in the process diagram are:

Functional: the definition of problem, the basis for agreement between buyer and seller, and the description of product that will be developed;

Design: for each software entity, the overall scheme of meeting its requirement, and the description of how it will be built;

Product: the definition of each software entity at the time of formal test;

Operational: the definition of each software entity at any time during its service life;
As one of the quality assurance activities, configuration audit can be carried out to enable accuracy, integrity and completeness of products to be demonstrated and invalid claims to be disallowed. For example, to finalise the production baseline, the configuration manager should:

- compare source/object in different media
- correlate with design baseline
- compare design and source
- establish back-up
- verify with user
- demonstrate performance under test conditions

4. Measurement

Paragraph 6.4.2 Process Measurement of the ISO 9000-3 standard states that "The supplier should have quantitative measures of the quality of the development and delivery process. These metrics should reflect:

a) how well the development process is carried out in terms of milestone and in-process quality objectives being met on schedule;

b) how effective the development process is at reducing the probability that fault are introduced or that any faults introduced go undetected." [12]

To fulfil this requirement, besides measurement of in-process quality objectives such as number of faults found during testing, we must be able to establish a baseline for comparison, and for process improvement. One such baseline is the size of the software project.

4.1 Software Project Estimation

Many software project estimation models are based on historical data and they take the form:

\[ d = f(v_i) \]

where \( d \) is one of a number of estimated values (e.g. effort, cost, project duration) and \( v_i \) are selected independent parameters (e.g. estimated line of code LOC).

For example, the Basic model of COstructive COst MOdel (COCOMO) [9] for software project estimation is as follow:

\[ E \text{ (effort in man-month)} = a \text{ (KLOC)} e^b \]
\[ D \text{ (development time in month)} = c \text{ (E)} e^d \]

where \( a, b, c, d \) are constants, and KLOC is thousand lines of code.

The limitation of models based on estimated/delivered lines of code is obvious. The implementation-oriented size metric is not always meaningful to the end-user and reliable estimation of KLOC at early stages of software development process is difficult, if not impossible. Function Point Analysis (FPA) was first introduced by Allan Albrecht [4,1] to help
measure application development productivity in business information systems. The decomposition techniques list, classify, count, and weigh functions provided by the application. An adjustment factor is determined by considering fourteen general application attributes which cover factors such as transaction volume, ease of change etc. The adjusted function point can be used as a measure of project size to estimate development effort as well as productivity and process quality. Examples of those measures include Functions Points / Work Hours, and Defects / Function Point.

Besides software size estimation, software quality metrics must be used in order to make assessments throughout the software life cycle as to whether the quality objectives are being met. Examples of software products and their corresponding quality metrics throughout the software development life cycle are shown in the following diagram.

![Measurement of Software Process and Product](image)

**4.2 Requirement Specification**

Due to amplification of specification errors and the cost to detect and rectify these errors in later phases of the development cycle, the quality of specification is most crucial in the software process. Review techniques such as walkthrough and inspection are used for the purpose of error detection and to verify compliance of specific software products. Quantitative measurement of quality objectives are usually in the form of checklist. Documentation relevant to the system would be thoroughly researched to produce a list of requirements to be followed through during review of the system. The use of metrics based on checklists does not eliminate the need for subjective judgement in quality evaluations.
Alternatively, specifications written in formal specification language may solve the problem. Formal methods use mathematical notations, such as logic and set theory, to describe system specifications and software design together with techniques of validation and verification based on mathematics. The use of mathematically based formal specification languages and proof systems enables errors to be discovered much earlier and provides an unambiguous basis for system design. The combination of precision and the ability to argue rigorously assure the quality of specification which contributes to the development of quality software.

4.3 Design

Use as an example of design quality metric, Design Stability Metric [10] measures program stability as the inverse of a weighted sum of the complexity of the modules to which a variable definition change may ripple. For a module, the potential ripple effect is defined to be the total number of assumptions made by other modules, which

1) invoke the module
2) are invoked by the module
3) share global data or files with the module.

Thus the design stability of a program is defined as the total potential ripple effect of all its modules which constitute the key component for overall maintainability measure.

4.4 Implementation

To measure program quality, a numbers of metrics have been proposed. The Software Science Model [5] proposed by Maurice Halstead applies the scientific method to the properties and structure of computer programs. Based on the count of operators and operands in the program, the model can be applied to predict the number of program bugs, program length and effort to implement the program. A graph-theoretic complexity measure [7] proposed by Thomas J. McCabe is used to measure program testability and maintainability. The cyclomatic complexity of programs provides a quantitative basis for modularization and helps to identify software modules that will be difficult to test and maintain.

4.5 Testing

During the testing phase of the life cycle, software reliability metrics will be used to measure the operational characteristics of the program such as program correctness, and failure count. For the two classes of reliability model, the time-dependent class treats programming errors as an inherent program property. Assumptions are made about this property and used to model error behaviour as a function of time. For example
Given the following:

\[ P\{t < t_1 \leq t + \Delta t \mid t_1 > t\} = Z(t) \Delta t \]

where \( P\{t < t_1 \leq t + \Delta t \mid t_1 > t\} \) is the probability that a failure will surface and be discovered in time interval \( t \) to \( t + \Delta t \) given that no failure has occurred until time \( t \), and \( Z(t) \) is the failure rate (hazard functions). Software reliability \( R(t) \) is the probability that no errors have occurred within a given time period, and is given by

\[ R(t) = \exp\left(-\int_0^t Z(t) \, dt\right) \]

In the case of time-independent models, errors are handled in a statistical manner. Tests which specifically check for error conditions are carefully defined using a priori knowledge of the program. These tests form an experiment which is conducted on the software system under test. Experimental results are used to calculate figures of merit. For example, in Nelson’s Model [10], program reliability \( (R) \) is measured as

\[ R = 1 - \frac{n}{N} \]

where

\[ n = \text{number of inputs with execution failures} \]
\[ N = \text{total number of inputs}. \]

And can be estimated by

\[ R = 1 - \frac{n_\ast}{n} \]

where

\[ n = \text{number of random samples of inputs} \]
\[ n_\ast = \text{number of sample inputs which cause a failure} \]

To evaluate adequacy of test data, cyclomatic complexity can be used as criteria to determine the coverage of a program logic using white-box testing technique. Another way to evaluate adequacy of test data is to calculate the mutation score of test data in mutation testing during which test data is applied to the program \( P \) being tested and its mutants \( M(P) \). Let \( E(P) \) represent the set of equivalent mutants of \( P \) i.e. they will be indistinguishable from \( P \) under all test data, \( DM(P,D) \) represents the set of mutants which will return results which differ from the results which \( P \) delivers on \( D \). Mutation score is defined to be the fraction of nonequivalent mutants of \( P \) which are undistinguished by the test set \( D \). Thus, if \( m,e, dm \) represent the number of elements in \( M(P) \), \( E(P) \), and \( DM(P) \) respectively, then the mutation score of \( D \) and \( P \) is defined as: \( ms(P,D) = \frac{dm}{m-e} \) which is in the interval \([0,1]\), where a high score indicates that \( D \) is being close to adequate.
V. INCORPORATING SQA INTO A COMPUTING CURRICULUM

The Bachelor of Science with Honours in Computer Studies (BScCS) at the City Polytechnic of Hong Kong was started in October 1987. The programme of study focuses on the design and construction of systems in a broad area of information processing. A 'sandwich' mode of study is adopted, that is, after two years of initial study, students spend one year working full-time in the computing field, and then return to the Polytechnic for their final year of study.

The basic unit of teaching at the City Polytechnic is the module, which typically consists of 3 to 4 contact hours per week. Contact hours can be a combination of lectures, tutorials and laboratory sessions. In the BScCS, software quality assurance is covered in varying degrees at all stages of the programme. Specifically, it is an integral part of modules with a significant emphasis on the design and/or implementation of software systems. These modules include:

- **Year (1)**
  - Foundations of Computer Science
  - File Processing

- **Year (2)**
  - Software Engineering
  - Programming Languages
  - Operating Systems and Computer Architecture
  - Information System Analysis and Design

- **Year (4)**
  - Advanced Software Engineering
  - Information Systems Management
  - Project

Software Engineering, in year 2, provides a thorough coverage of the software life cycle, prevailing software design methodologies, and a comprehensive introduction to the principles of software quality. The SQA theme is elaborated in Advanced Software Engineering, in the final year of study, with emphasis on more detailed aspects of configuration management and software quality measurement, together with an introduction to a formal specification language. In the other modules, aspects of SQA are covered in a 'soft' manner - practices of good software engineering principles, which help to improve software quality, are introduced, reiterated, and encouraged. This blending of formal and informal approaches, realised throughout the curriculum, helps students to develop a much-needed awareness of, and belief in, an appropriate quality culture.
Furthermore, the industrial placement year enables students to have first-hand experience with how software quality assurance is practised in industry. Through their participation in software development and maintenance projects, they have ample opportunities to observe and evaluate the operations of their organisations and to discuss these with their academic supervisors. With an increasing number of information processing companies falling in line with ISO standards, this interaction between academia and industry will play an increasingly important role in fostering SQA in the future.

VI. CONCLUSION

At the current state of implementation of the degree programme, SQA topics are given the greatest coverage in the final-year module Advanced Software Engineering. Module feedback from students has been very positive. Students have found the material practical, in that they are able to apply the concepts and techniques in software development projects. On the other hand, they have some difficulties in writing up some of the SQA quality documents prescribed by the framework. This is understandable, in view of the scale of student project, which is much smaller than that of samples used as references in international standards on software documentation.

As part of the process of continual development of the degree programme, it would appear desirable, even necessary, to enhance the SQA elements in the curriculum. The effectiveness of the framework for teaching SQA can be increased by making selected SQA procedures a requirement in modules involving significant software development. A natural target for this would be the final-year project, which, in almost all cases, entails design and implementation of non-trivial software systems. Also, it would be beneficial to make SQA observations and evaluations a compulsory section of the industrial placement report, to enable students to gain better insights in an industrial context.

It is worth mentioning, as a concluding remark, that future developments, as is the existing framework, will be based on ideas drawn from published international standards, such as the ISO 9000 Standards, to ensure continuity and professional relevance.
REFERENCES


