24 Quality management

Objectives

The objectives of this chapter are to introduce software quality management and to describe specific quality management activities. When you have read this chapter, you will:

- understand the quality management process and the key process activities of quality assurance, quality planning and quality control;
- understand the importance of standards in the quality management process;
- understand the notion of a software metric and the differences between predictor metrics and control metrics;
- understand how measurement may be helpful in assessing some quality attributes and the current limitations of software measurement.

Contents

Achieving a high level of product or service quality is the objective of most organisations. It is no longer acceptable to deliver poor quality products and then repair problems and deficiencies after they have been delivered to the customer. In this respect, software is the same as any other manufactured product such as cars, televisions or computers.

However, software quality is a complex concept that cannot be defined in a simple way. Classically, the notion of quality has been that the developed product should meet its specification (Crosby, 1979). In an ideal world, this definition should apply to all products but, for software systems, there are problems:

1. The specification should be oriented towards the characteristics of the product that the customer wants. However, the development organisation may also have requirements (such as maintainability requirements) which are not included in the specification.
2. We do not know how to specify certain quality characteristics (e.g. maintainability) in an unambiguous way.
3. As I discussed in Part 1, which covered requirements engineering, it is very difficult to write complete software specifications. Therefore, although a software product may conform to its specification, users may not consider it to be a high-quality product.

Obviously, efforts should be made to improve specifications but currently we have to accept that these will be imperfect. We should recognise the problems with existing specifications and put procedures in place to improve quality within the constraints imposed by an imperfect specification. In particular, software attributes such as maintainability, portability or efficiency may be critical quality attributes that are not specified explicitly but which affect the perceived quality of the system. I discuss these quality attributes in section 24.2 which covers quality planning.

The responsibility of quality managers in an organisation is to ensure that the required level of product quality is achieved. In principle, quality management simply involves defining procedures and standards which should be used during software development and checking that these are followed by all engineers. In practice, however, there is more to quality management than this.

Good quality managers aim to develop a 'quality culture' where everyone responsible for product development is committed to achieving a high level of product quality. They encourage teams to take responsibility for the quality of their work and to develop new approaches to quality improvement. While standards and procedures are the basis of quality management, experienced quality managers recognise that there are intangible aspects to software quality (elegance, readability, etc.) which cannot be embodied in standards. They support people who are interested in these intangible aspects of quality and encourage professional behaviour in all team members.

Software quality management can be structured into three principal activities:

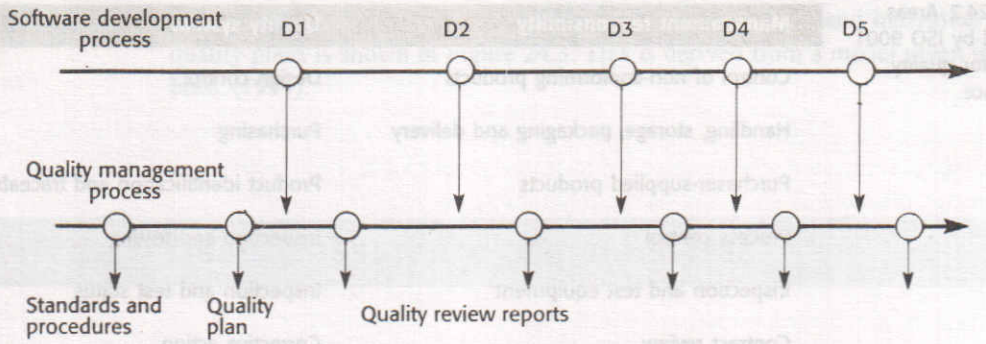


Figure 24.1 Quality management and software development

1. *Quality assurance* The establishment of a framework of organisational procedures and standards which lead to high-quality software.
2. *Quality planning* The selection of appropriate procedures and standards from this framework and the adaptation of these for a specific software project.
3. *Quality control* The definition and enactment of processes which ensure that the project quality procedures and standards are followed by the software development team.

Quality management provides an independent check on the software development process. The deliverables from the software process are input to the quality management process and are checked to ensure that they are consistent with organisational standards and goals (Figure 24.1). As the quality assurance and control team should be independent, they can take an objective view of the process and can report problems and difficulties to senior management in the organisation.

Quality management should be separated from project management so that quality is not compromised by management responsibilities for project budget and schedule. An independent team should be responsible for quality management and should report to management above the project manager level. The quality management team should not be associated with any particular development group but should take organisation-wide responsibility for quality management.

An international standard that can be used in the development of a quality management system in all industries is called ISO 9000. ISO 9000 is a set of standards that can be applied to a range of organisations from manufacturing through to service industries. ISO 9001 is the most general of these standards and applies to organisations concerned with the quality process in organisations which design, develop and maintain products. A supporting document (ISO 9000-3) interprets ISO 9000 for software development. Several books describing the ISO 9000 standard are available (Johnson, 1993; Oskarsson and Glass, 1995; Peach, 1996).

ISO 9001 is a generic model of a quality process. It describes various aspects of that process and defines which standards and procedures should exist within an organisation. As it is not industry-specific, these are not defined in detail. Within any specific organisation, a set of appropriate quality processes should be defined and documented in an organisational quality manual.

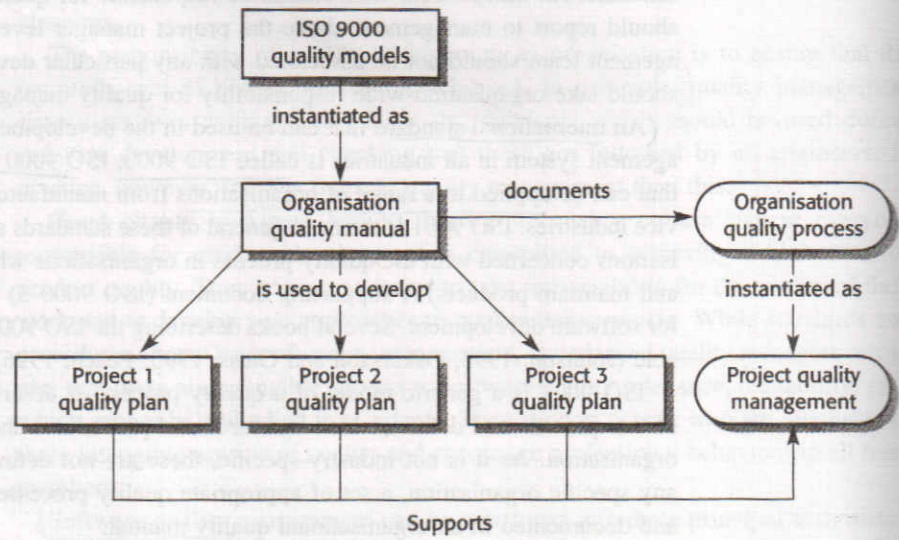
Figure 24.2 Areas covered by ISO 9001 model for quality assurance

Management responsibility	Quality system
Control of non-conforming products	Design control
Handling, storage, packaging and delivery	Purchasing
Purchaser-supplied products	Product identification and traceability
Process control	Inspection and testing
Inspection and test equipment	Inspection and test status
Contract review	Corrective action
Document control	Quality records
Internal quality audits	Training
Servicing	Statistical techniques

Figure 24.2 shows the areas which are covered in ISO 9001. I do not have space here to discuss this standard in any depth. Ince (1994) and Oskarrson and Glass (1995) give a more detailed account of how the standard can be used to develop software quality management processes.

The quality assurance procedures in an organisation are documented in a quality manual which defines the quality process. In some countries, bodies exist which will certify that the quality process as expressed in the quality manual conforms to ISO 9001. Increasingly, customers look for ISO 9000 certification in a supplier as an indicator of how seriously that supplier takes quality.

Figure 24.3 ISO 9000 and quality management



The relationship between ISO 9000, the quality manual and individual project quality plans is shown in Figure 24.3. This is derived from a model given in Ince's book (1994).

24.1 Quality assurance and standards

Quality assurance (QA) activities define a framework for achieving software quality. The QA process involves defining or selecting standards that should be applied to the software development process or software product. These standards may be embedded in procedures or processes which are applied during development. Processes may be supported by tools that embed knowledge of the quality standards.

There are two types of standard that may be established as part of the quality assurance process:

1. *Product standards* These are standards that apply to the software product being developed. They include document standards such as the structure of the requirements document which should be produced, documentation standards such as a standard comment header for an object class definition and coding standards which define how a programming language should be used.
2. *Process standards* These are standards that define the processes which should be followed during software development. They may include definitions of specification, design and validation processes and a description of the documents which must be generated in the course of these processes.

There is a very close relationship between product and process standards. The product standards apply to the outputs of the software process and, in many cases, the process standards include specific process activities that ensure that product standards are followed. I discuss the important relationship between process and product quality in section 24.1.2.

There are a number of reasons why software standards are important:

1. They provide an encapsulation of best, or at least most appropriate, practice. This knowledge is often only acquired after a great deal of trial and error. Building it into a standard avoids the repetition of past mistakes. Standards capture wisdom that is of value to the organisation.
2. They provide a framework around which the quality assurance process may be implemented. Given that standards encapsulate best practice, quality control simply involves ensuring that standards have been properly followed.

Figure 24.4 Product and process standards

Product standards	Process standards
Design review form	Design review conduct
Requirements document structure	Submission of documents to CM
Procedure header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

3. They assist in continuity where work carried out by one person is taken up and continued by another. Standards ensure that all engineers within an organisation adopt the same practices. Consequently, learning effort when starting new work is reduced.

The development of software engineering project standards is a difficult and time-consuming process. National and international bodies such as the US DoD, ANSI, BSI, NATO and the IEEE have been active in the production of standards. These are general standards that can be applied across a range of projects. Bodies such as NATO and other defence organisations may require that their own standards are followed in software contracts.

National and international standards have been developed covering software engineering terminology, programming languages such as Ada and C++, notations such as charting symbols, procedures for deriving and writing software requirements, quality assurance procedures and software verification and validation processes (IEEE, 1994).

Quality assurance teams who are developing standards should normally base organisational standards on national and international standards. Using these standards as a starting point, the quality assurance team should draw up a standards 'handbook'. This should define the standards that are appropriate for their organisation. Examples of standards that might be included in such a handbook are shown in Figure 24.4.

Software engineers sometimes consider standards to be bureaucratic and irrelevant to the technical activity of software development. This is particularly likely when the standards require tedious form-filling and work recording. Although they usually agree about the general need for standards, engineers often find good reasons why standards are not necessarily appropriate to their particular project.

To avoid these problems, quality managers who set the standards need to be adequately resourced and should take the following steps:

1. Involve software engineers in the development of product standards. They should understand the motivation behind the development of the standard and be

committed to these standards. The standards document should not simply state a standard to be followed but should include a rationale of why particular standardisation decisions have been made.

2. Review and modify standards regularly to reflect changing technologies. Once standards are developed they tend to be enshrined in a company standards handbook and there is often a reluctance to change them. A standards handbook is essential but it should evolve with changing circumstances and technology.

3. Provide software tools to support standards wherever possible. Clerical standards are the cause of many complaints because of the tedious work involved in implementing them. If tool support is available, there is not a great deal of additional effort involved in development to the standards.

Process standards may cause difficulties if an impractical process is imposed on the development team. Such standards are often simply guidelines which must be sympathetically interpreted by individual project managers. There is no point in prescribing a particular way of working if it is inappropriate for a project or project team. Each project manager should therefore have the authority to modify process standards according to individual circumstances. However, standards that relate to product quality and the post-delivery process should only be changed after careful consideration.

The project manager and the quality manager can avoid the problems of inappropriate standards by careful quality planning. They should decide which of the standards in the handbook should be used without change, which should be modified and which should be ignored. New standards may have to be created in response to a particular project requirement. For example, standards for formal specifications may be required if these have not been used in previous projects. These new standards must be allowed to evolve during the project.

24.1.1 Documentation standards

Documentation standards in a software project are particularly important as documents are the only tangible way of representing the software and the software process. Standardised documents have a consistent appearance, structure and quality and should therefore be easier to read and understand.

There are three types of documentation standards:

1. *Documentation process standards* These standards define the process which should be followed for document production.
2. *Document standards* These are standards that govern the structure and presentation of documents.
3. *Document interchange standards* These are standards that ensure that all electronic copies of documents are compatible.

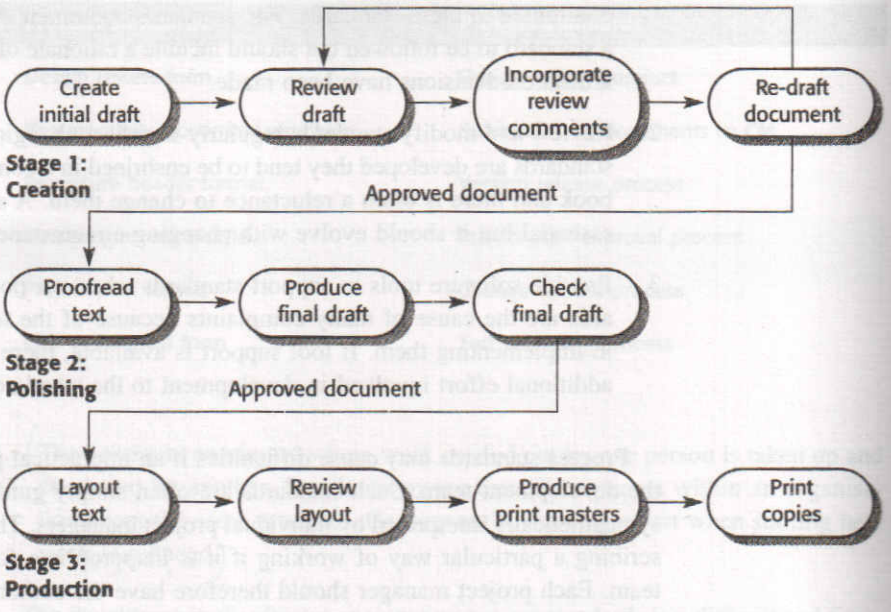


Figure 24.5 A document production process including quality checks

Process standards define the process used to produce documents. This means defining the procedures involved in document development and the software tools used for document production. Checking and refinement procedures which ensure that high-quality documents are produced should also be defined.

Document process quality standards must be flexible and must be able to cope with all types of document. For working papers or memos, there is no need for explicit quality checking. However, where documents are formal documents used for further development or are released to customers, a formal quality process should be adopted. Figure 24.5 is a model of one possible process.

Drafting, checking, revising and redrafting is an iterative process. It should continue until a document of acceptable quality is produced. The acceptable quality level depends on the document type and the potential readers of the document.

Document standards should apply to all documents produced in the course of the software development. Documents should have a consistent style and appearance and documents of the same type should have a consistent structure. Although document standards should be adapted to the needs of a specific project, it is good practice for the same 'house style' to be used in all of the documents produced by an organisation.

Examples of document standards which may be developed are:

1. *Document identification standards* As large systems projects may produce thousands of documents, each document must be uniquely identified. For formal documents, this identifier may be the formal identifier defined by the configuration manager. For informal documents, the style of the document identifier should be defined by the project manager.

2. *Document structure standards* Each class of document produced during a software project should follow some standard structure. Structure standards should define the sections to be included and should specify the conventions used for page numbering, page header and footer information, and section and subsection numbering.

3. *Document presentation standards* Document presentation standards define a 'house style' for documents and they contribute significantly to document consistency. They include the definition of fonts and styles used in the document, the use of logos and company names, the use of colour to highlight document structure, etc.

4. *Document update standards* As a document evolves to reflect changes in the system, a consistent way of indicating document changes should be used. You can use different colours of cover to indicate a new document version and change bars in the margin to indicate modified or added paragraphs.

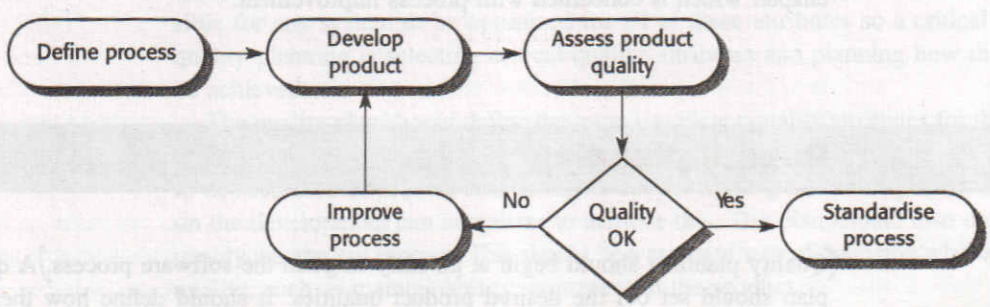
Document interchange standards are important as electronic copies of documents are interchanged. The use of interchange standards allows documents to be transferred electronically and re-created in their original form.

Assuming that the use of standard tools is mandated in the process standards, interchange standards define the conventions for using these tools. Examples of interchange standards include the use of an agreed standard macro set if a text formatting system is used for document production or the use of a standard style sheet if a word processor is used. Interchange standards may also limit the fonts and text styles used because of differing printer and display capabilities.

24.1.2 Process and product quality

An underlying assumption of quality management is that the quality of the development process directly affects the quality of delivered products. This assumption is derived from manufacturing systems where product quality is intimately related to the production process. Indeed, in automated mass production systems, once an acceptable level of process quality has been attained, product quality naturally follows. Figure 24.6 illustrates this approach to quality assurance.

Figure 24.6 Process-based quality



Process quality is particularly important in software development. The reason for this is that it is difficult to measure software attributes, such as maintainability, without using the software for a long period. Quality improvement focuses on identifying good quality products, examining the processes used to develop these products, then generalising these processes so that they may be applied across a range of projects. However, the relationship between software process and software product quality is complex. Changing the process does not always lead to improved product quality.

There is a clear link between process and product quality in manufacturing because the process is relatively easy to standardise and monitor. Once manufacturing systems are calibrated, they can be run again and again to output high-quality products. Software is not manufactured but is designed. As software development is a creative rather than a mechanical process, the influence of individual skills and experience is significant. External factors, such as the novelty of an application or commercial pressure for an early product release, also affect product quality irrespective of the process used.

Nevertheless, process quality has a significant influence on the quality of the software. Process quality management involves:

1. Defining process standards such as how reviews should be conducted, when reviews should be held, etc.
2. Monitoring the development process to ensure that the standards are being followed.
3. Reporting the software process to project management and to the buyer of the software.

A danger of process-based quality assurance is that the prescribed process may be inappropriate for the type of software which is being developed. For example, process quality standards may specify that a specification must be complete and approved before implementation can begin. However, some systems may require prototyping which involves program implementation. The quality team may suggest that this prototyping should not be carried out because its quality cannot be monitored. In such situations, senior management must intervene to ensure that the quality process supports rather than hinders product development.

I return to the relationships between process and product quality in the next chapter which is concerned with process improvement.

24.2 Quality planning

Quality planning should begin at an early stage in the software process. A quality plan should set out the desired product qualities. It should define how these are

Figure 24.7 Software quality attributes

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

to be assessed. It therefore defines what 'high quality' software actually means. Without such a definition, different engineers may work in an opposing way so that different product attributes are optimised. The result of the quality planning process is a project quality plan.

The quality plan should select those organisational standards that are appropriate to a particular product and development process. New standards may have to be defined if the project uses new methods and tools. Humphrey (1989), in his classic book on software management, suggests an outline structure for a quality plan. This includes:

1. *Product introduction* A description of the product, its intended market and the quality expectations for the product.
2. *Product plans* The critical release dates and responsibilities for the product along with plans for distribution and product servicing.
3. *Process descriptions* The development and service processes which should be used for product development and management.
4. *Quality goals* The quality goals and plans for the product, including an identification and justification of critical product quality attributes.
5. *Risks and risk management* The key risks which might affect product quality and the actions to address these risks.

When writing quality plans, you should try to keep them as short as possible. If the document is too long, engineers will not read it and this will defeat the purpose of producing a quality plan.

There is a wide range of potential software quality attributes (Figure 24.7) that should be considered during the quality planning process. In general, it is not possible for any system to be optimised for all of these attributes so a critical part of quality planning is selecting critical quality attributes and planning how these can be achieved.

The quality plan should define the most significant quality attributes for the product being developed. It may be that efficiency is paramount and other factors have to be sacrificed to achieve this. If this is set out in the plan, the engineers working on the development can cooperate to achieve this. The plan should also define the quality assessment process. This should be a standard way of assessing whether some quality, such as maintainability, is present in the product.

24.3 Quality control

Quality control involves overseeing the software development process to ensure that quality assurance procedures and standards are being followed. As discussed earlier in the chapter (see Figure 24.1), the deliverables from the software process are checked against the defined project standards in the quality control process.

The quality control process has its own set of procedures and reports that must be used during software development. These procedures should be straightforward and easily understood by the engineers developing the software.

There are two complementary approaches to quality control:

1. Quality reviews where the software, its documentation and the processes used to produce that software are reviewed by a group of people. They are responsible for checking that the project standards have been followed and that software and documents conform to these standards. Deviations from the standards are noted and brought to the attention of project management.
2. Automated software assessment where the software and the documents which are produced are processed by some program and compared to the standards which apply to that particular development project. This automated assessment may involve a quantitative measurement of some software attributes. I discuss software measurement and metrics in section 24.4.

24.3.1 Quality reviews

Reviews are the most widely used method of validating the quality of a process or product. They involve a group of people examining part or all of a software process, system or its associated documentation to discover potential problems. The conclusions of the review are formally recorded and passed to the author or whoever is responsible for correcting the discovered problems.

Figure 24.8 briefly describes several different types of review. Program inspections have already been covered in Chapter 19. Progress reviews are part of the management process as discussed in Chapter 4. In this chapter, I concentrate on reviews as part of the quality management process. The different review processes have much in common and I have already described the process of setting up a review in Chapter 19.

The remit of the review team is to detect errors and inconsistencies and point them out to the designer or document author. Reviews are document-based but are not limited to specifications, designs or code. Documents such as process models, test plans, configuration management procedures, process standards and user manuals may all be reviewed.

The review team should include those project members who can make an effective contribution. For example, if a sub-system design is being reviewed, designers of related sub-systems should be included in the review team. They may bring important

Figure 24.8 Types of review

Review type	Principal purpose
Design or program inspections	To detect detailed errors in the requirements, design or code. The review should be driven by a checklist of possible errors.
Progress reviews	To provide information for management about the overall progress of the project. This is both a process and a product review and is concerned with costs, plans and schedules.
Quality reviews	To carry out a technical analysis of product components or documentation to find mismatches between the specification and the component design, code or documentation and to ensure that defined quality standards have been followed.

insights into sub-system interfaces which could be missed if the sub-system is considered in isolation.

The review team should have a core of 3–4 people who are selected as principal reviewers. One member should be a senior designer who can take the responsibility for making significant technical decisions. The principal reviewers may invite other project members to contribute to the review. They may not be involved in reviewing the whole document. Rather, they concentrate on those parts which affect their work. Alternatively, the review team may circulate the document being reviewed and ask for written comments from other project members.

Documents to be reviewed must be distributed well in advance of the review to allow reviewers time to read and understand them. Although this delay can disrupt the development process, reviewing is ineffective if the review team have not properly understood the documents before the review takes place.

The review itself should be relatively short (two hours at most). The author of the document being reviewed should 'walk through' the document with the review team. One team member should chair the review and another should formally record all review decisions. During the review, the chair is responsible for ensuring that all written comments are considered. On completion of the review, the actions are noted and forms recording the comments and actions are signed by the designer and the review chair. These are then filed as part of the formal project documentation. If only minor problems are discovered, a further review may be unnecessary. The chairman is responsible for ensuring that the required changes are made. If major changes are necessary, a follow-on review may be arranged.

24.4 Software measurement and metrics

Software measurement is concerned with deriving a numeric value for some attribute of a software product or a software process. By comparing these values

to each other and to standards which apply across an organisation, it is possible to draw conclusions about the quality of software or software processes. For example, say an organisation plans to introduce a new software testing tool. Before introducing the tool, the number of software defects discovered in a given time may be recorded; after introducing the tool, this process is repeated. If more defects are discovered in the same amount of time after introducing the tool then it would appear that it provides useful support for the software validation process.

A number of large companies such as Hewlett-Packard (Grady, 1993) and AT&T (Barnard and Price, 1994) have introduced metrics programmes and are using collected metrics in their quality management processes. Most of the focus has been on collecting metrics on program defects and the verification and validation processes. Offen and Jeffrey (1997) and Hall and Fenton (1997) discuss the introduction of such programmes in industry. The ami handbook (Pulford *et al.*, 1996) gives detailed advice on measurement and using measurement for process improvement.

However, the use of systematic software measurement and metrics is still relatively uncommon. There is a reluctance to introduce measurement because the benefits are unclear. One reason for this is that, in many companies, the software processes used are still poorly organised and are not sufficiently mature to make use of measurements. Another reason is that there are no standards for metrics and hence limited support for data collection and analysis. Most companies will not be prepared to introduce measurement until such standards and tools are available.

A software metric is any type of measurement which relates to a software system, process or related documentation. Examples are measures of the size of a product in lines of code, the Fog index (Gunning, 1962) which is a measure of the readability of a passage of written text, the number of reported faults in a delivered software product and the number of person-days required to develop a system component.

Metrics may be either control metrics or predictor metrics. Control metrics are usually associated with software processes; predictor metrics are associated with software products. Examples of control or process metrics are the average effort and time required to repair reported defects. Examples of predictor metrics are the cyclomatic complexity of a module, the average length of identifiers in a program and the number of attributes and operations associated with objects in a design. Both control and predictor metrics may influence management decision making as shown in Figure 24.9. I cover process metrics and their role in process improvement in Chapter 25. The focus here is on measurement to predict software product quality.

It is often impossible to measure software quality attributes directly. Attributes such as maintainability, complexity and understandability are affected by many different factors and there are no straightforward metrics for them. Rather, we have to measure some internal attribute of the software (such as its size) and assume that a relationship exists between what we can measure and what we want to know. Ideally, there should be a clear and validated relationship between the internal and the external software attributes.

Figure 24.10 shows some external quality attributes which might be of interest and internal attributes which can be measured and which might be related to the

Figure 24.9 Predictor and control metrics

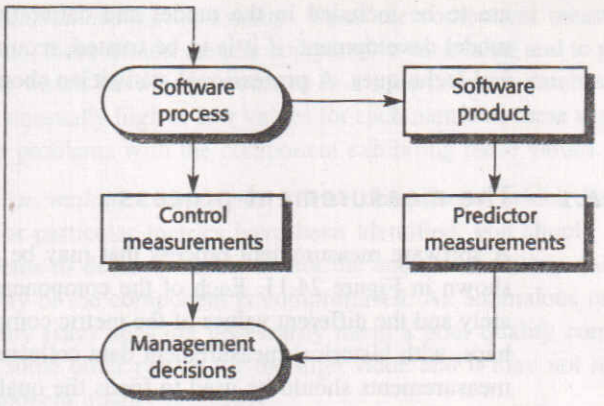
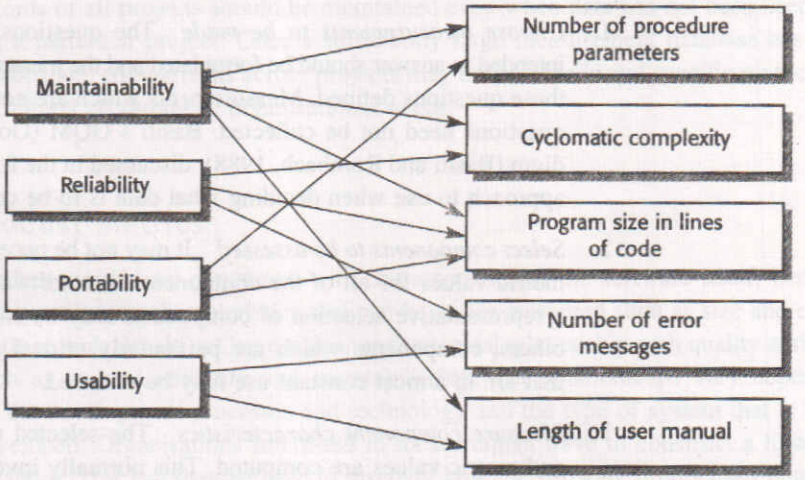


Figure 24.10 Relationships between internal and external software attributes



external attribute. The diagram suggests that there may be a relationship between external and internal attributes but it does not say what that relationship is. If the measure of the internal attribute is to be a useful predictor of the external software characteristic, three conditions must hold (Kitchenham, 1990):

1. The internal attribute must be measured accurately.
2. A relationship must exist between what we can measure and the external behavioural attribute.
3. This relationship is understood, has been validated and can be expressed in terms of a formula or model.

Model formulation involves identifying the functional form of the model (linear, exponential, etc.) by analysis of collected data, identifying the parameters which

are to be included in the model and calibrating these using existing data. Such model development, if it is to be trusted, requires significant experience in statistical techniques. A professional statistician should be involved in the process.

24.4.1 The measurement process

A software measurement process that may be part of a quality control process is shown in Figure 24.11. Each of the components of the system is analysed separately and the different values of the metric compared both with each other and, perhaps, with historical measurement data collected on previous projects. Anomalous measurements should be used to focus the quality assurance effort on components that may have quality problems.

The key stages in this process are:

1. *Choose measurements to be made* The questions that the measurement is intended to answer should be formulated and the measurements required to answer these questions defined. Measurements which are not directly relevant to these questions need not be collected. Basili's GQM (Goal-Question-Metric) paradigm (Basili and Rombach, 1988), discussed in the following chapter, is a good approach to use when deciding what data is to be collected.

2. *Select components to be assessed* It may not be necessary or desirable to assess metric values for all of the components in a software system. In some cases, a representative selection of components may be chosen for measurement. In others, components which are particularly critical such as core components that are in almost constant use may be assessed.

3. *Measure component characteristics* The selected components are measured and metric values are computed. This normally involves processing the component representation (design, code, etc.) using an automated data collection tool. This may be specially written or may already be incorporated in CASE tools that are used in an organisation.

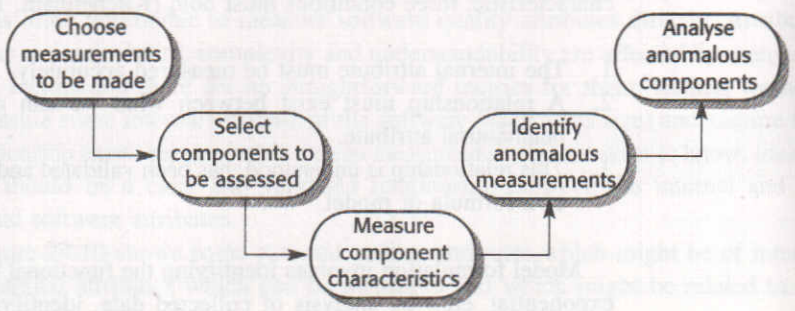


Figure 24.11 The process of product measurement

4. *Identify anomalous measurements* Once the component measurements have been made, these should then be compared to each other and to previous measurements which have been recorded in a measurement database. You should look for unusually high or low values for each metric as these suggest that there could be problems with the component exhibiting these values.

5. *Analyse anomalous components* Once components which have anomalous values for particular metrics have been identified, you should examine these components to decide whether or not the anomalous metric values mean that the quality of the component is compromised. An anomalous metric value for complexity (say) does not necessarily mean a poor quality component. There may be some other reason for the high value and it may not mean that there are component quality problems.

Collected data should be maintained as an organisational resource and historical records of all projects should be maintained even when data has not been used during a particular project. Once a sufficiently large measurement database has been established, comparisons across projects may then be made and specific metrics can be refined according to organisational needs.

24.4.2 Product metrics

Product metrics are concerned with characteristics of the software itself. Unfortunately, software characteristics that can be easily measured such as size and cyclomatic complexity do not have a clear and universal relationship with quality attributes such as understandability and maintainability. The relationships vary depending on the development processes and technology and the type of system that is being developed. Organisations interested in measurement have to construct a historical database. This can then be used to discover how the software product attributes are related to the qualities of interest to the organisation.

Product metrics fall into two classes:

1. Dynamic metrics which are collected by measurements made of a program in execution.
2. Static metrics which are collected by measurements made of the system representations such as the design, program or documentation.

These different types of metric are related to different quality attributes. Dynamic metrics help to assess the efficiency and the reliability of a program whereas static metrics help to assess the complexity, understandability and maintainability of a software system.

Dynamic metrics are usually fairly closely related to software quality attributes. It is relatively easy to measure the execution time required for particular functions and to assess the time required to start up a system. These relate directly to the

Figure 24.12 Software product metrics

Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions that call some other function (say X). Fan-out is the number of functions which are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a program component, the more complex and error-prone that component is likely to be.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. The computation of cyclomatic complexity is covered in Chapter 20.
Length of identifiers	This is a measure of the average length of distinct identifiers in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and are potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value for the Fog index, the more difficult the document may be to understand.

system's efficiency. Similarly, the number of system failures and the type of failure can be logged and related directly to the reliability of the software as discussed in Chapter 16.

Static metrics, on the other hand, have an indirect relationship with quality attributes. A large number of these metrics have been proposed and experiments carried out to try to derive and validate the relationships between these metrics and system complexity, understandability and maintainability. Figure 24.12 describes several static metrics which have been used for assessing quality attributes. Of these, program or component length and control complexity seem to be the most reliable predictors of understandability, system complexity and maintainability.

Since the early 1990s, there have been a number of studies of object-oriented metrics. Some of these have been derived from the older metrics shown in Figure 24.12 but others are unique to object-oriented systems. A number of object-oriented metrics are explained in Figure 24.13. These metrics are less mature than the metrics in Figure 24.12 which are intended for function-oriented designs so their usefulness as predictor metrics is still being established.

Figure 24.13 Object-oriented metrics

Object-oriented metric	Description
Depth of inheritance tree	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design as, potentially, many different object classes have to be understood to understand the object classes at the leaves of the tree.
Method fan-in/fan-out	This is directly related to fan-in and fan-out as described above and means essentially the same thing. However, it may be appropriate to make a distinction between calls from other methods within the object and calls from external methods.
Weighted methods per class	This is the number of methods included in a class weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1 and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be more difficult to understand. They may not be logically cohesive so cannot be reused effectively as superclasses in an inheritance tree.
Number of overriding operations	These are the number of operations in a superclass which are overridden in a subclass. A high value for this metric indicates that the superclass used may not be an appropriate parent for the subclass.

The specific metrics that are relevant depend on the project, the goals of the quality management team and the type of software that is being developed. All of the metrics shown in Figures 24.12 and 24.13 may be useful in some situations. Equally, however, there will be other situations where they are inappropriate. When introducing software measurement as part of the quality management process, organisations should experiment to discover the most appropriate metrics for their needs.

24.4.3 Analysis of measurements

One of the problems with collecting quantitative data about software and software projects is understanding what that data really means. It is easy to misinterpret data and to make inferences that are incorrect. Measurements must be carefully analysed to understand what they really mean.

To illustrate how collected data can be interpreted in different ways consider the following scenario:

A manager decides to monitor the number of change requests submitted by customers based on an assumption that there is a relationship between these change

requests and product usability and suitability. The higher the number of change requests, the less the software meets the needs of the customer.

Processing change requests and changing the software is expensive. The organisation therefore decides to modify its process to increase customer satisfaction and, at the same time, reduce the costs of change. It is intended that the process changes will result in better products and fewer change requests.

Process changes are initiated which involve more customer involvement in the software design process. Beta-testing of all products is introduced and customer-requested modifications are incorporated in the delivered product. New versions of products, developed with this modified process, are delivered. In some cases, the number of change requests is reduced; in others, it is increased. The manager is baffled and cannot assess the effects of the process changes on the product quality.

To understand why this kind of thing can happen, you have to understand why change requests are made. One reason is that the delivered software does not do what customers want it to do. Another possibility is that the software is very good and is widely and heavily used, sometimes for purposes for which it was not originally designed. Because there are so many people using it, it is natural that more change requests are generated.

A third possibility is that the company producing the software is responsive to customers' change requests. Customers are therefore satisfied with the service they receive. They generate a lot of change requests because they know that these requests will be taken seriously. Their suggestions will probably be incorporated in later versions of the software.

The number of change requests might decrease because the process changes have been effective and have made the software more usable and suitable. Alternatively, the number might have decreased because the product has lost market share to a rival product. There are consequently fewer product users. The number of change requests might increase because there are more users, because the beta-testing process has convinced users that the company is willing to make changes or because the beta-test sites were not typical of most usage of the program.

To analyse the change request data, we do not simply need to know the number of change requests. We need to know who made the request, how they use the software and why the request was made. We need to know if there are external factors such as modifications to the change request procedure or market changes which might have an effect. With this information, it is then possible to find out if the process changes have been effective in increasing product quality.

This illustrates that interpreting quantitative data about a product or a process is an uncertain process. Processes and products that are being measured are not insulated from their environment and changes to that environment may make comparisons of data invalid. Quantitative data about human activities cannot always be taken at face value. Underlying reasons that might account for the measured value should be investigated.

KEY POINTS

- Software quality management is concerned with ensuring that software has a low number of defects and that it reaches the required standards of maintainability, reliability, portability, etc. Quality management activities include quality assurance that sets the standards for software development, quality planning and quality control that checks the software against the defined standards.
- An organisational quality manual should document a set of quality assurance procedures. This may be based on the generic model suggested in the ISO 9000 standards.
- Software standards are important to quality assurance as they represent an identification of 'best practice'. The quality control process is concerned with checking that the software process and the software being developed conform to these standards.
- Reviews of the software process deliverables are the most widely used technique for assessing quality.
- Software measurement can be used to gather quantitative data about software and the software process. The values of the software metrics which are collected may be used to make inferences about product and process quality.
- Product quality metrics are particularly valuable for highlighting anomalous components which may have quality problems. These components should then be analysed in more detail.
- There are no standardised and universally applicable software metrics. Organisations must select metrics and analyse measurements based on local knowledge and circumstances.

FURTHER READING

'Making sense of measurement for small organisations'. This is an interesting article about the practical application of metrics. It makes the point that all uses of metrics have to take their context into account. (K. Kautz, *IEEE Software*, March/April 1999.)

ISO 9000 and Software Quality Assurance. This is a readable introduction to ISO 9000 and its relevance to software quality assurance. Its structure is based on the quality standard with a chapter on each topic. (D. Ince, 1994, McGraw-Hill.)

IEEE Software, March 1997. This is a special issue on software measurement which has particularly useful articles on introducing a measurement programme.

A quantitative approach to software management: The ami handbook. This is an excellent 'how-to' guide which discusses how to introduce a measurement programme and use the results for process improvement. (K. Pulford, A. Kuntzmann-Combelles and S. Shirlaw, 1996, Addison-Wesley.)

EXERCISES

24.1 Explain why a high-quality software process should lead to high-quality software products. Discuss possible problems with this system of quality management.

24.2 What are the stages involved in the review of a software design?

24.3 Discuss the assessment of software quality according to the quality attributes shown in Figure 24.7. You should consider each attribute in turn and explain how it might be assessed.

24.4 Design an electronic form that may be used to record review comments and which could be used to mail comments electronically to reviewers.

24.5 Briefly describe possible standards which might be used for:

- the use of control constructs in C, C++ or Java;
- reports which might be submitted for a term project in a university;
- the process of making and approving changes to a program (see Chapter 29);
- the process of purchasing and installing a new computer.

24.6 Assume you work for an organisation that develops database products for microcomputer systems. This organisation is interested in quantifying its software development. Write a report suggesting appropriate metrics and suggest how these can be collected.

24.7 Explain why design metrics are, by themselves, an inadequate method of predicting design quality.

24.8 Consult the literature and find other design quality metrics that have been suggested apart from those discussed here. Consider these metrics in detail and assess whether they are likely to be of real value.

24.9 Do software standards stifle technological innovation?

24.10 A colleague who is a very good programmer produces software with a low number of defects but consistently ignores organisational quality standards. How should the managers in the organisation react to this behaviour?