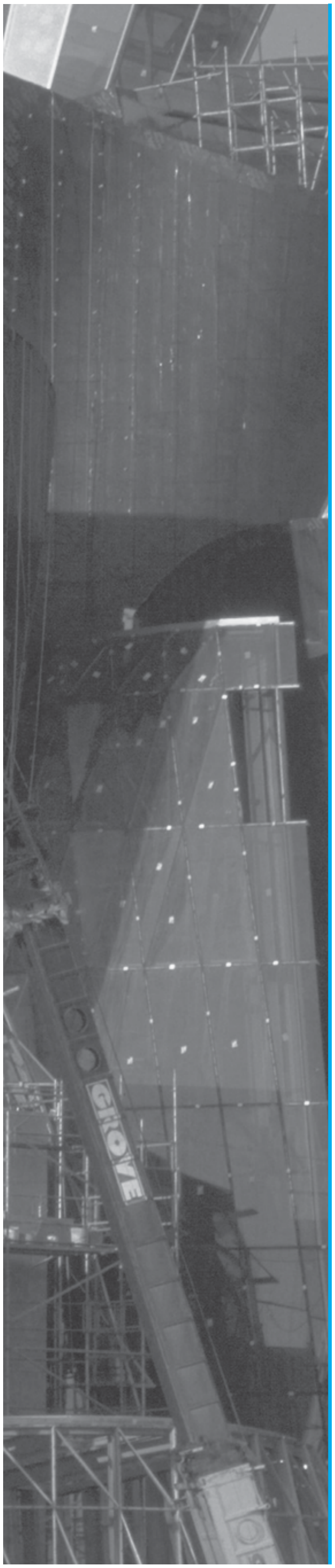

24

Quality management

Objectives

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

- have been introduced to the quality management process and know why quality planning is important;
- understand that software quality is affected by the software development process used;
- be aware of the importance of standards in the quality management process and know how standards are used in quality assurance;
- understand how reviews and inspections are used as a mechanism for software quality assurance;
- understand how measurement may be helpful in assessing some software quality attributes and the current limitations of software measurement.

Contents

Problems with software quality were initially discovered in the 1960s with the development of the first large software systems, and continued to plague software engineering throughout the 20th century. Delivered software was slow and unreliable, difficult to maintain and hard to reuse. Dissatisfaction with this situation led to the adoption of formal techniques of software quality management, which have been developed from methods used in the manufacturing industry. These quality management techniques, in conjunction with new software technologies and better software testing, have led to significant improvements in the general level of software quality.

Software quality management for software systems has three principal concerns:

1. At the organizational level, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high-quality software. This means that the quality management team should take responsibility for defining the software development processes to be used and standards that should apply to the software and related documentation, including the system requirements, design, and code.

2. At the project level, quality management involves the application of specific quality processes, checking that these planned processes have been followed, and ensuring that the project outputs are conformant with the standards that are applicable to that project.

3. Quality management at the project level is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

The terms ‘quality assurance’ and ‘quality control’ are widely used in manufacturing industry. Quality assurance (QA) is the definition of processes and standards that should lead to high-quality products and the introduction of quality processes into the manufacturing process. Quality control is the application of these quality processes to weed out products that are not of the required level of quality.

In the software industry, different companies and industry sectors interpret quality assurance and quality control in different ways. Sometimes, quality assurance simply means the definition of procedures, processes, and standards that are aimed at ensuring that software quality is achieved. In other cases, quality assurance also includes all configuration management, verification, and validation activities that are applied after a product has been handed over by a development team. In this chapter, I use the term ‘quality assurance’ to include verification and validation and the processes of checking that quality procedures have been properly applied. I have avoided the term ‘quality control’ as this term is not widely used in the software industry.

The QA team in most companies is responsible for managing the release testing process. As I discussed in Chapter 8, this means that they manage the testing of the software before it is released to customers. They are responsible for checking that the system tests provide coverage of the requirements and that proper records are maintained of the testing process. As I have covered release testing in Chapter 8, I do not cover this aspect of quality assurance here.

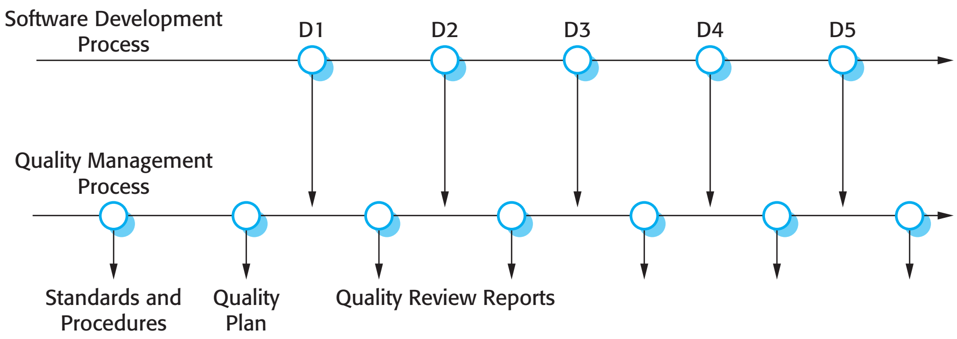

Figure 24.1 Quality management and software development

Quality management provides an independent check on the software development process. The quality management process checks the project deliverables to ensure that they are consistent with organizational standards and goals (Figure 24.1). The QA team should be independent from the development team so that they can take an objective view of the software. This allows them to report on software quality without being influenced by software development issues.

Ideally, the quality management team should not be associated with any particular development group, but should rather have organization-wide responsibility for quality management. They should be independent and report to management above the project manager level. The reason for this is that project managers have to maintain the project budget and schedule. If problems arise, they may be tempted to compromise on product quality so that they meet their schedule. An independent quality management team ensures that the organizational goals of quality are not compromised by short-term budget and schedule considerations. In smaller companies, however, this is practically impossible. Quality management and software development are inevitably intertwined with people having both development and quality responsibilities.

Quality planning is the process of developing a quality plan for a project. The quality plan should set out the desired software qualities and describe how these are to be assessed. It therefore defines what ‘high-quality’ software actually means for a particular system. Without this definition, engineers may make different and sometimes conflicting assumptions about which product attributes reflect the most important quality characteristics. Formalized quality planning is an integral part of plan-based development processes. Agile methods, however, adopt a less formal approach to quality management.

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 and standards that 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 that might affect product quality and the actions to be taken to address these risks.

Quality plans, which are developed as part of the general project planning process, differ in detail depending on the size and the type of system that is being developed. However, when writing quality plans, you should try to keep them as short as possible. If the document is too long, people will not read it and this will defeat the purpose of producing the quality plan.

Some people think that software quality can be achieved through prescriptive processes that are based around organizational standards and associated quality procedures that check that these standards are followed by the software development team. Their argument is that standards embody good software engineering practice and that following this good practice will lead to high-quality products. In practice, however, I think that there is much more to quality management than standards and the associated bureaucracy to ensure that these have been followed.

Standards and processes are important but quality managers should also aim to develop a ‘quality culture’ where everyone responsible for software development is committed to achieving a high level of product quality. They should encourage teams to take responsibility for the quality of their work and to develop new approaches to quality improvement. Although standards and procedures are the basis of quality management, good quality managers recognize that there are intangible aspects to software quality (elegance, readability, etc.) that cannot be embodied in standards. They should support people who are interested in the intangible aspects of quality and encourage professional behavior in all team members.

Formalized quality management is particularly important for teams that are developing large, long-lifetime systems that take several years to develop. Quality documentation is a record of what has been done by each subgroup in the project. It helps people check that important tasks have not been forgotten or that one group has not made incorrect assumptions about what other groups have done. The quality documentation is also a means of communication over the lifetime of a system. It allows the groups responsible for system evolution to trace the tests and checks that have been implemented by the development team.

For smaller systems, quality management is still important but a more informal approach can be adopted. Not as much paperwork is needed because a small development team can communicate informally. The key quality issue for small systems development is establishing a quality culture and ensuring that all team members have a positive approach to software quality.

24.1 Software quality

The fundamentals of quality management were established by manufacturing industry in a drive to improve the quality of the products that were being made. As part of this, they developed a definition of 'quality', which was based on conformance with a detailed product specification (Crosby, 1979) and the notion of tolerances. The underlying assumption was that products could be completely specified and procedures could be established that could check a manufactured product against its specification. Of course, products will never exactly meet a specification so some tolerance was allowed. If the product was 'almost right', it was classed as acceptable.

Software quality is not directly comparable with quality in manufacturing. The idea of tolerances is not applicable to digital systems and, for the following reasons, it may be impossible to come to an objective conclusion about whether or not a software system meets its specification:

1. As I discussed in Chapter 4, which covered requirements engineering, it is difficult to write complete and unambiguous software specifications. Software developers and customers may interpret the requirements in different ways and it may be impossible to reach agreement on whether or not software conforms to its specification.

2. Specifications usually integrate requirements from several classes of stakeholders. These requirements are inevitably a compromise and may not include the requirements of all stakeholder groups. The excluded stakeholders may therefore perceive the system as a poor quality system, even though it implements the agreed requirements.

3. It is impossible to measure certain quality characteristics (e.g., maintainability) directly and so they cannot be specified in an unambiguous way. I discuss the difficulties of measurement in Section 24.4.

Because of these problems, the assessment of software quality is a subjective process where the quality management team has to use their judgment to decide if an acceptable level of quality has been achieved. The quality management team has to consider whether or not the software is fit for its intended purpose. This involves answering questions about the system's characteristics. For example:

1. Have programming and documentation standards been followed in the development process?

2. Has the software been properly tested?

3. Is the software sufficiently dependable to be put into use?

4. Is the performance of the software acceptable for normal use?

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

Figure 24.2 Software quality attributes

5. Is the software usable?

6. Is the software well structured and understandable?

There is a general assumption in software quality management that the system will be tested against its requirements. The judgment on whether or not it delivers the required functionality should be based on the results of these tests. Therefore, the QA team should review the tests that have been developed and examine the test records to check that testing has been properly carried out. In some organizations, the quality management team is responsible for system testing but, sometimes, a separate system testing group is made responsible for this.

The subjective quality of a software system is largely based on its non-functional characteristics. This reflects practical user experience—if the software's functionality is not what is expected, then users will often just work around this and find other ways to do what they want to do. However, if the software is unreliable or too slow, then it is practically impossible for them to achieve their goals.

Therefore software quality is not just about whether the software functionality has been correctly implemented, but also depends on non-functional system attributes. Boehm, et al. (1978) suggested that there were 15 important software quality attributes, as shown in Figure 24.2. These attributes relate to the software dependability, usability, efficiency, and maintainability. As I have discussed in Chapter 11, I believe that dependability attributes are usually the most important quality attributes of a system. However, the software's performance is also very important. Users will reject software that is too slow.

It is not possible for any system to be optimized for all of these attributes—for example, improving robustness may lead to loss of performance. The quality plan should therefore define the most important quality attributes for the software that is being developed. It may be that efficiency is critical and other factors have to be sacrificed to achieve this. If you have stated this in the quality plan, the engineers working on the development can cooperate to achieve this. The plan should also include a definition of the quality assessment process. This should be an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

An assumption that underlies software quality management is that the quality of software is directly related to the quality of the software development process. This again comes from manufacturing systems where product quality is intimately related

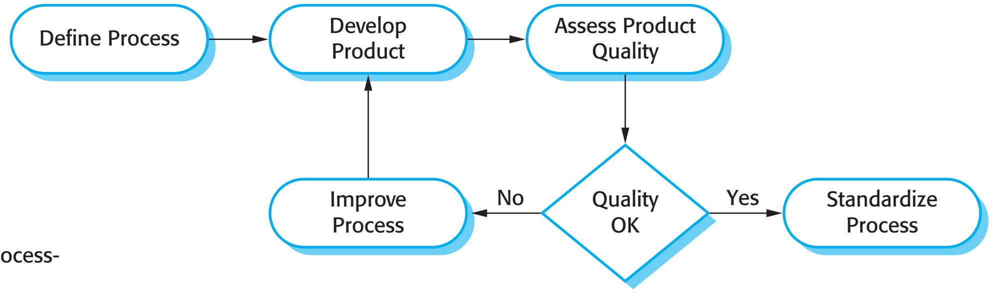

Figure 24.3 Process-based quality

to the production process. A manufacturing process involves configuring, setting up, and operating the machines involved in the process. Once the machines are operating correctly, product quality naturally follows. You measure the quality of the product and change the process until you achieve the quality level that you need. Figure 24.3 illustrates this process-based approach to achieving product quality.

There is a clear link between process and product quality in manufacturing because the process is relatively easy to standardize and monitor. Once manufacturing systems are calibrated, they can be run again and again to output high-quality products. However, software is not manufactured—it is designed. In software development, therefore, the relationship between process quality and product quality is more complex. Software development is a creative rather than a mechanical process, so 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.

There is no doubt that the development process used has a significant influence on the quality of the software and that good processes are more likely to lead to good quality software. Process quality management and improvement can lead to fewer defects in the software being developed. However, it is difficult to assess software quality attributes, such as maintainability, without using the software for a long period. Consequently, it is hard to tell how process characteristics influence these attributes. Furthermore, because of the role of design and creativity in the software process, process standardization can sometimes stifle creativity, which leads to poorer rather than better quality software.

24.2 Software standards

Software standards play a very important role in software quality management. As I have discussed, an important part of quality assurance is the definition or selection of standards that should apply to the software development process or software product. As part of this QA process, tools and methods to support the use of these standards may also be chosen. Once standards have been selected for use, project-specific

Documentation standards

Project documents are a tangible way of describing the different representations of a software system (requirements, UML, code, etc.) and its production process. Documentation standards define the organization of different types of documents as well as the document format. They are important because they make it easier to check that important material has not been omitted from documents and ensure that project documents have a common 'look and feel'. Standards may be developed for the process of writing documents, for the documents themselves, and for document exchange.

http://www.SoftwareEngineering-9.com/Web/QualityMan/docstandards.html

processes have to be defined to monitor the use of the standards and check that they have been followed.

Software standards are important for three reasons:

1. Standards capture wisdom that is of value to the organization. They are based on knowledge about the best or most appropriate practice for the company. This knowledge is often only acquired after a great deal of trial and error. Building it into a standard helps the company reuse this experience and avoid previous mistakes.
2. Standards provide a framework for defining what 'quality' means in a particular setting. As I have discussed, software quality is subjective, and by using standards you establish a basis for deciding if a required level of quality has been achieved. Of course, this depends on setting standards that reflect user expectations for software dependability, usability, and performance.
3. Standards assist continuity when work carried out by one person is taken up and continued by another. Standards ensure that all engineers within an organization adopt the same practices. Consequently, the learning effort required when starting new work is reduced.

There are two related types of software engineering standard that may be defined and used in software quality management:

1. *Product standards* These apply to the software product being developed. They include document standards, such as the structure of requirements documents, 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 define the processes that should be followed during software development. They should encapsulate good development practice. Process standards may include definitions of specification, design and validation processes, process support tools, and a description of the documents that should be written during these processes.

Product standards	Process standards
Design review form	Design review conduct
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Figure 24.4 Product and process standards

Standards have to deliver value, in the form of increased product quality. There is no point in defining standards that are expensive in terms of time and effort to apply that only lead to marginal improvements in quality. Product standards have to be designed so that they can be applied and checked in a cost-effective way, and process standards should include the definition of processes that check that product standards have been followed.

The development of international software engineering standards is usually a prolonged process where those interested in the standard meet, produce drafts for comment, and finally agree on the standard. National and international bodies such as the U.S. DoD, ANSI, BSI, NATO, and the IEEE support the production of standards. These are general standards that can be applied across a range of projects. Bodies such as NATO and other defense organizations may require that their own standards be used in the development contracts that they place with software companies.

National and international standards have been developed covering software engineering terminology, programming languages such as Java and C++, notations such as charting symbols, procedures for deriving and writing software requirements, quality assurance procedures, and software verification and validation processes (IEEE, 2003). More specialized standards, such as IEC 61508 (IEC, 1998), have been developed for safety and security-critical systems.

Quality management teams that are developing standards for a company should normally base these company standards on national and international standards. Using international standards as a starting point, the quality assurance team should draw up a standards ‘handbook’. This should define the standards that are needed by their organization. Examples of standards that could be included in such a handbook are shown in Figure 24.4.

Software engineers sometimes consider standards to be overprescriptive and not really relevant to the technical activity of software development. This is particularly likely when project standards require tedious documentation 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 minimize dissatisfaction and to encourage buy-in to standards, quality managers who set the standards should therefore take the following steps:

1. *Involve software engineers in the selection of product standards* If developers understand why standards have been selected, they are more likely to be committed to these standards. Ideally, the standards document should not just set out the standard to be followed, but should also include commentary explaining why standardization decisions have been made.
2. *Review and modify standards regularly to reflect changing technologies* Standards are expensive to develop and they tend to be enshrined in a company standards handbook. Because of the costs and discussion required, there is often a reluctance to change them. A standards handbook is essential but it should evolve to reflect changing circumstances and technology.
3. *Provide software tools to support standards* Developers often find standards to be a bugbear when conformance to them involves tedious manual work that could be done by a software tool. If tool support is available, very little effort is required to follow the software development standards. For example, document standards can be implemented using word processor styles.

Different types of software need different development processes so standards have to be adaptable. 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 have the authority to modify process standards according to individual circumstances. However, when changes are made, it is important to ensure that these changes do not lead to a loss of product quality. This will affect an organization's relationship with its customers and will probably lead to increased project costs.

The project manager and the quality manager can avoid the problems of inappropriate standards by careful quality planning early in the project. They should decide which of the organizational standards should be used without change, which should be modified, and which should be ignored. New standards may have to be created in response to customer or project requirements. For example, standards for formal specifications may be required if these have not been used in previous projects.

24.2.1 The ISO 9001 standards framework

There is an international set of standards that can be used in the development of quality management systems in all industries, called ISO 9000. ISO 9000 standards can be applied to a range of organizations from manufacturing through to service industries. ISO 9001, the most general of these standards, applies to organizations that design, develop, and maintain products, including software. The ISO 9001 standard was originally developed in 1987, with its most recent revision in 2008.

The ISO 9001 standard is not itself a standard for software development but is a framework for developing software standards. It sets out general quality principles,

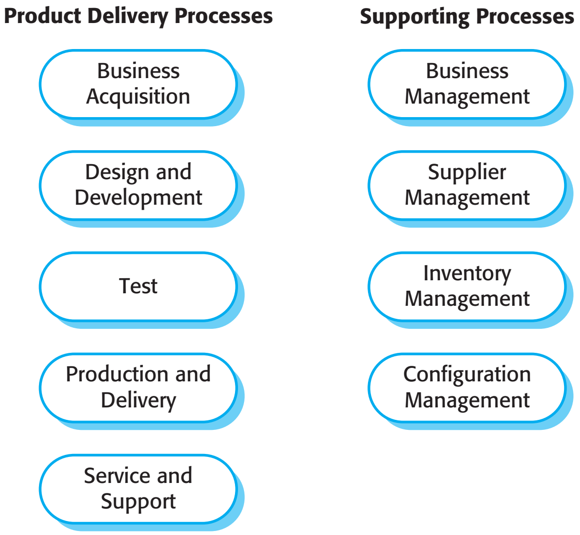

Figure 24.5 ISO 9001 core processes

describes quality processes in general, and lays out the organizational standards and procedures that should be defined. These should be documented in an organizational quality manual.

The major revision of the ISO 9001 standard in 2000 reoriented the standard around nine core processes (Figure 24.5). If an organization is to be ISO 9001 conformant, it must document how its processes relate to these core processes. It must also define and maintain records that demonstrate that the defined organizational processes have been followed. The company quality manual should describe the relevant processes and the process data that has to be collected and maintained.

The ISO 9001 standard does not define or prescribe the specific quality processes that should be used in a company. To be conformant with ISO 9001, a company must have defined the types of process shown in Figure 24.5 and have procedures in place that demonstrate that its quality processes are being followed. This allows flexibility across industrial sectors and company sizes. Quality standards can be defined that are appropriate for the type of software being developed. Small companies can have unbureaucratic processes and still be ISO 9001 compliant. However, this flexibility means that you cannot make assumptions about the similarities or differences between the processes in different ISO 9001–compliant companies. Some companies may have very rigid quality processes that keep detailed records, whereas others may be much less formal, with minimal additional documentation.

The relationships between ISO 9001, organizational quality manuals, and individual project quality plans are shown in Figure 24.6. This diagram has been derived from a model given by Ince (1994), who explains how the general ISO 9001 standard can be used as a basis for software quality management processes. Bamford and Dielbler (2003) explain how the later ISO 9001: 2000 standard can be applied in software companies.

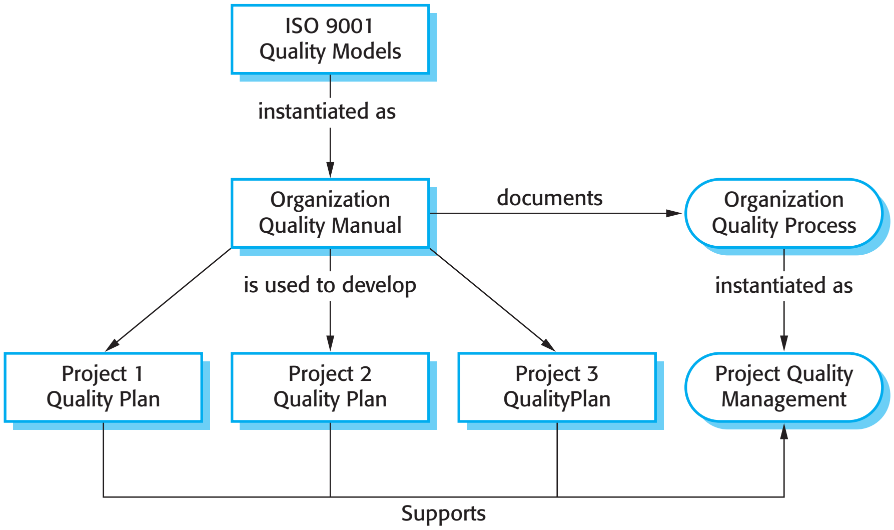

Figure 24.6 ISO 9001 and quality management

Some software customers demand that their suppliers should be ISO 9001 certified. The customers can then be confident that the software development company has an approved quality management system in place. Independent accreditation authorities examine the quality management processes and process documentation and decide if these processes cover all of the areas specified in ISO 9001. If so, they certify that a company's quality processes, as defined in the quality manual, conform to the ISO 9001 standard.

Some people think that ISO 9001 certification means that the quality of the software produced by certified companies will be better than that from uncertified companies. This is not necessarily true. The ISO 9001 standard focuses on ensuring that the organization has quality management procedures in place and it follows these procedures. There is no guarantee that ISO 9001 certified companies use the best software development practices or that their processes lead to high-quality software.

For example, a company could define test coverage standards specifying that all methods in objects must be called at least once. Unfortunately, this standard can be met by incomplete software testing, which does not run tests with different method parameters. So long as the defined testing procedures were followed and records kept of the testing carried out, the company could be ISO 9001 certified. The ISO 9001 certification defines quality to be the conformance to standards, and takes no account of the quality as experienced by users of the software.

Agile methods, which avoid documentation and focus on the code being developed, have little in common with the formal quality processes that are discussed in ISO 9001. There has been some work done on reconciling these approaches (Stalhane and Hanssen, 2008), but the agile development community is fundamentally opposed to what they see as the bureaucratic overhead of standards conformance. For this reason, companies that use agile development methods are rarely concerned with ISO 9001 certification.

24.3 Reviews and inspections

Reviews and inspections are QA activities that check the quality of project deliverables. This involves examining the software, its documentation and records of the process to discover errors and omissions and to see if quality standards have been followed. As I discussed in Chapters 8 and 15, reviews and inspections are used alongside program testing as part of the general process of software verification and validation.

During a review, a group of people examine the software and its associated documentation, looking for potential problems and non-conformance with standards. The review team makes informed judgments about the level of quality of a system or project deliverable. Project managers may then use these assessments to make planning decisions and allocate resources to the development process.

Quality reviews are based on documents that have been produced during the software development process. As well as software specifications, designs, or code, process models, test plans, configuration management procedures, process standards, and user manuals may all be reviewed. The review should check the consistency and completeness of the documents or code under review and make sure that quality standards have been followed.

However, reviews are not just about checking conformance to standards. They are also used to help discover problems and omissions in the software or project documentation. The conclusions of the review should be formally recorded as part of the quality management process. If problems have been discovered, the reviewers' comments should be passed to the author of the software or whoever is responsible for correcting errors or omissions.

The purpose of reviews and inspections is to improve software quality, not to assess the performance of people in the development team. Reviewing is a public process of error detection, compared with the more private component-testing process. Inevitably, mistakes that are made by individuals are revealed to the whole programming team. To ensure that all developers engage constructively with the review process, project managers have to be sensitive to individual concerns. They must develop a working culture that provides support without blame when errors are discovered.

Although a quality review provides information for management about the software being developed, quality reviews are not the same as management progress reviews. As I discussed in Chapter 23, progress reviews compare the actual progress in a software project against the planned progress. Their prime concern is whether or not the project will deliver useful software on time and on budget. Progress reviews take external factors into account, and changed circumstances may mean that software under development is no longer required or has to be radically changed.

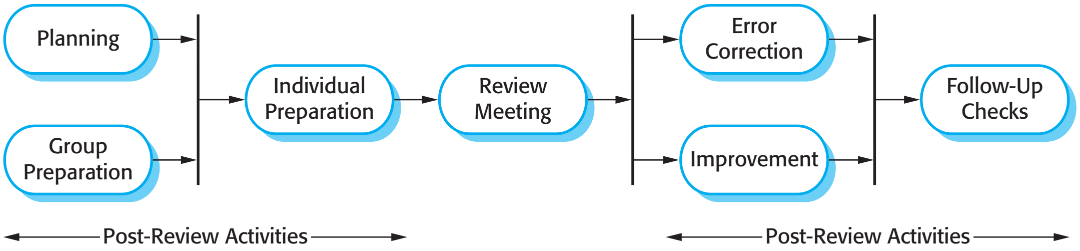

Figure 24.7 The software review process

Projects that have developed high-quality software may have to be canceled because of changes to the business or its operating environment.

24.3.1 The review process

Although there are many variations in the details of reviews, the review process (Figure 24.7) is normally structured into three phases:

1. *Pre-review activities* These are preparatory activities that are essential for the review to be effective. Typically, pre-review activities are concerned with review planning and review preparation. Review planning involves setting up a review team, arranging a time and place for the review, and distributing the documents to be reviewed. During review preparation, the team may meet to get an overview of the software to be reviewed. Individual review team members read and understand the software or documents and relevant standards. They work independently to find errors, omissions, and departures from standards. Reviewers may supply written comments on the software if they cannot attend the review meeting.

2. *The review meeting* During the review meeting, an author of the document or program being reviewed should 'walk through' the document with the review team. The review itself should be relatively short—two hours at most. One team member should chair the review and another should formally record all review decisions and actions to be taken. During the review, the chair is responsible for ensuring that all written comments are considered. The review chair should sign a record of comments and actions agreed during the review.

3. *Post-review activities* After a review meeting has finished, the issues and problems raised during the review must be addressed. This may involve fixing software bugs, refactoring software so that it conforms to quality standards, or rewriting documents. Sometimes, the problems discovered in a quality review are such that a management review is also necessary to decide if more resources should be made available to correct them. After changes have been made, the review chair may check that the review comments have all been taken into account. Sometimes, a further review will be required to check that the changes made cover all of the previous review comments.

Roles in the inspection process

When program inspection was first established in IBM (Fagan, 1976; Fagan, 1986), there were a number of formal roles defined for members of the inspection team. These included moderator, code reader, and scribe. Other users of inspections have modified these roles but it is generally accepted that an inspection should involve the code author, an inspector, and a scribe and should be chaired by a moderator.

http://www.SoftwareEngineering-9.com/Web/QualityMan/roles.html

Review teams should normally have a core of three to four people who are selected as principal reviewers. One member should be a senior designer who will take the responsibility for making significant technical decisions. The principal reviewers may invite other project members, such as the designers of related subsystems, to contribute to the review. They may not be involved in reviewing the whole document but should concentrate on those sections that affect their work. Alternatively, the review team may circulate the document and ask for written comments from a broad spectrum of project members. The project manager need not be involved in the review, unless problems are anticipated that require changes to the project plan.

The above review process relies on all members of a development team being colocated and available for a team meeting. However, project teams are now often distributed, sometimes across countries or continents, so it is often impractical for team members to meet in the same room. In such situations, document editing tools may be used to support the review process. Team members use these to annotate the document or software source code with comments. These comments are visible to other team members who may then approve or reject them. A phone discussion may only be required when disagreements between reviewers have to be resolved.

The review process in agile software development is usually informal. In Scrum, for example, there is a review meeting after each iteration of the software has been completed (a sprint review), where quality issues and problems may be discussed. In extreme programming, as I discuss in the next section, pair programming ensures that code is constantly being examined and reviewed by another team member. General quality issues are also considered at daily team meetings but XP relies on individuals taking the initiative to improve and refactor code. Agile approaches are not usually standards-driven, so issues of standards compliance are not usually considered.

The lack of formal quality procedures in agile methods means that there can be problems in using agile approaches in companies that have developed detailed quality management procedures. Quality reviews can slow down the pace of software development and they are best used within a plan-driven development process. In a plan-driven process, reviews can be planned and other work scheduled in parallel with them. This is impractical in agile approaches that focus single-mindedly on code development.

24.3.2 Program inspections

Program inspections are 'peer reviews' where team members collaborate to find bugs in the program that is being developed. As I discussed in Chapter 8, inspections may be part of the software verification and validation processes. They complement testing as they do not require the program to be executed. This means that incomplete versions of the system can be verified and that representations such as UML models can be checked. Gilb and Graham (1993) suggest that one of the most effective ways to use inspections is to review the test cases for a system. Inspections can discover problems with tests and so improve the effectiveness of these tests in detecting program bugs.

Program inspections involve team members from different backgrounds who make a careful, line-by-line review of the program source code. They look for defects and problems and describe these at an inspection meeting. Defects may be logical errors, anomalies in the code that might indicate an erroneous condition or features that have been omitted from the code. The review team examines the design models or the program code in detail and highlights anomalies and problems for repair.

During an inspection, a checklist of common programming errors is often used to focus the search for bugs. This checklist may be based on examples from books or from knowledge of defects that are common in a particular application domain. You use different checklists for different programming languages because each language has its own characteristic errors. Humphrey (1989), in a comprehensive discussion of inspections, gives a number of examples of inspection checklists.

Possible checks that might be made during the inspection process are shown in Figure 24.8. Gilb and Graham (1993) emphasize that each organization should develop its own inspection checklist based on local standards and practices. These checklists should be regularly updated, as new types of defects are found. The items in the checklist vary according to programming language because of the different levels of checking that are possible at compile-time. For example, a Java compiler checks that functions have the correct number of parameters; a C compiler does not.

Most companies that have introduced inspections have found that they are very effective in finding bugs. Fagan (1986) reported that more than 60 percent of the errors in a program can be detected using informal program inspections. Mills et al. (1987) suggest that a more formal approach to inspection, based on correctness arguments, can detect more than 90% of the errors in a program. McConnell (2004) compares unit testing, where the defect detection rate is about 25%, with inspections, where the defect detection rate was 60%. He also describes a number of case studies including an example where the introduction of peer reviews led to a 14% increase in productivity and a 90% decrease in program defects.

In spite of their well-publicized cost effectiveness, many software development companies are reluctant to use inspections or peer reviews. Software engineers with experience of program testing are sometimes unwilling to accept that inspections can

Fault class	Inspection check
Data faults	• Are all program variables initialized before their values are used? • Have all constants been named? • Should the upper bound of arrays be equal to the size of the array or Size -1? • If character strings are used, is a delimiter explicitly assigned? • Is there any possibility of buffer overflow?
Control faults	• For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for? • If a break is required after each case in case statements, has it been included?
Input/output faults	• Are all input variables used? • Are all output variables assigned a value before they are output? • Can unexpected inputs cause corruption?
Interface faults	• Do all function and method calls have the correct number of parameters? • Do formal and actual parameter types match? • Are the parameters in the right order? • If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	• If a linked structure is modified, have all links been correctly reassigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly deallocated after it is no longer required?
Exception management faults	• Have all possible error conditions been taken into account?

Figure 24.8 An inspection checklist

be more effective for defect detection than testing. Managers may be suspicious because inspections require additional costs during design and development. They may not wish to take the risk that there will be no corresponding savings in program testing costs.

Agile processes rarely use formal inspection or peer review processes. Rather, they rely on team members cooperating to check each other's code, and informal guidelines, such as 'check before check-in', which suggest that programmers should check their own code. Extreme programming practitioners argue that pair programming is an effective substitute for inspection as this is, in effect, a continual inspection process. Two people look at every line of code and check it before it is accepted.

Pair programming leads to a deep knowledge of a program, as both programmers have to understand its working in detail to continue development. This depth of knowledge is sometimes difficult to achieve in other inspection processes and so pair programming can find bugs that sometimes would not be discovered in formal inspections. However, pair programming can also lead to mutual misunderstandings of requirements, where both members of the pair make the same mistake. Furthermore, pairs may be reluctant to look for errors because the pair does not want to slow down the progress of the project. The people involved cannot be as objective as an external inspection team and their ability to discover defects is likely to be compromised by their close working relationship.

24.4 Software measurement and metrics

Software measurement is concerned with deriving a numeric value or profile for an attribute of a software component, system, or process. By comparing these values to each other and to the standards that apply across an organization, you may be able to draw conclusions about the quality of software, or assess the effectiveness of software processes, tools, and methods.

For example, say an organization intends to introduce a new software-testing tool. Before introducing the tool, you record the number of software defects discovered in a given time. This is a baseline for assessing the effectiveness of the tool. After using the tool for some time, you repeat this process. If more defects have been found in the same amount of time, after the tool has been introduced, then you may decide that it provides useful support for the software validation process.

The long-term goal of software measurement is to use measurement in place of reviews to make judgments about software quality. Using software measurement, a system could ideally be assessed using a range of metrics and, from these measurements, a value for the quality of the system could be inferred. If the software had reached a required quality threshold, then it could be approved without review. When appropriate, the measurement tools might also highlight areas of the software that could be improved. However, we are still quite a long way from this ideal situation and, there are no signs that automated quality assessment will become a reality in the foreseeable future.

A software metric is a characteristic of a software system, system documentation, or development process that can be objectively measured. Examples of metrics include 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.

Software metrics may be either control metrics or predictor metrics. As the names imply, control metrics support process management, and predictor metrics help you predict characteristics of the software. Control metrics are usually associated with software processes. Examples of control or process metrics are the average effort and the time required to repair reported defects. Predictor metrics are associated with the software itself and are sometimes known as ‘product metrics’. Examples of predictor metrics are the cyclomatic complexity of a module (discussed in Chapter 8),

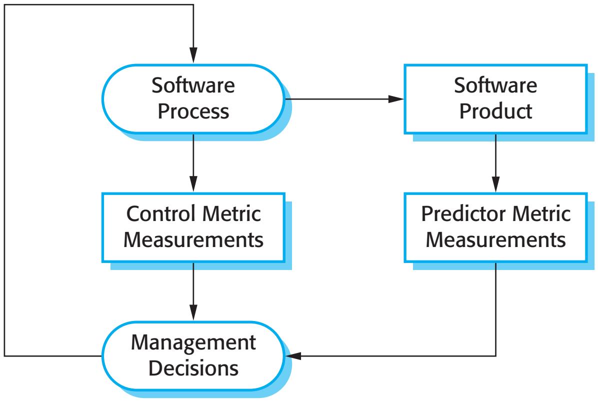

Figure 24.9 Predictor and control measurements

the average length of identifiers in a program, and the number of attributes and operations associated with object classes in a design.

Both control and predictor metrics may influence management decision making, as shown in Figure 24.9. Managers use process measurements to decide if process changes should be made, and predictor metrics to help estimate the effort required to make software changes. In this chapter, I mostly discuss predictor metrics, whose values are assessed by analyzing the code of a software system. I discuss control metrics and how they are used in process improvement in Chapter 26.

There are two ways in which measurements of a software system may be used:

1. *To assign a value to system quality attributes* By measuring the characteristics of system components, such as their cyclomatic complexity, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.

2. *To identify the system components whose quality is substandard* Measurements can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

Unfortunately, it is difficult to make direct measurements of many of the software quality attributes shown in Figure 24.2. Quality attributes such as maintainability, understandability, and usability are external attributes that relate to how developers and users experience the software. They are affected by subjective factors, such as user experience and education, and they cannot therefore be measured objectively. To make a judgment about these attributes, you have to measure some internal attributes of the software (such as its size, complexity, etc.) and assume that these are related to the quality characteristics that you are concerned with.

Figure 24.10 shows some external software quality attributes and internal attributes that could, intuitively, be related to them. The diagram suggests that there may be relationships between external and internal attributes, but it does not say how

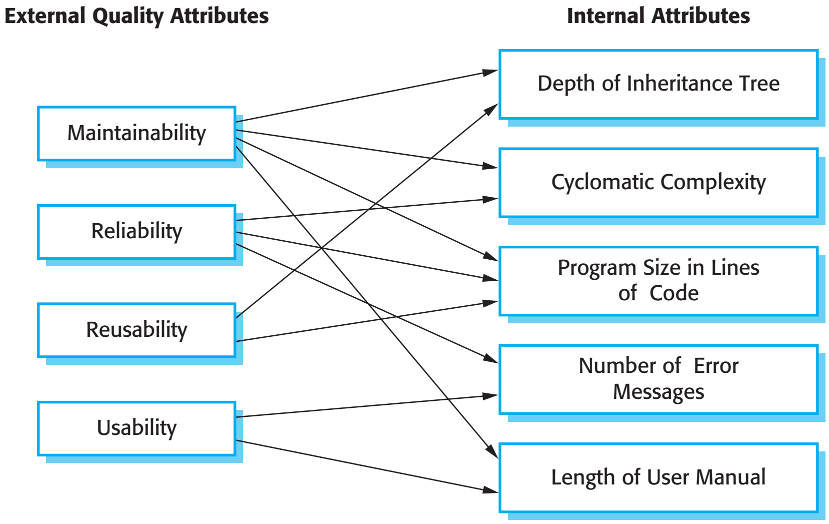

Figure 24.10 Relationships between internal and external software

these attributes are related. 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. This is not always straightforward and it may require special-purpose tools to make the measurements.
2. A relationship must exist between the attribute that can be measured and the external quality attribute that is of interest. That is, the value of the quality attribute must be related, in some way, to the value of the attribute than can be measured.
3. This relationship between the internal and external attributes must be understood, validated, and 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 that are to be included in the model, and calibrating these parameters using existing data.

Internal software attributes, such as the cyclomatic complexity of a component, are measured by using software tools that analyze the source code of the software. Open source tools are available that can be used to make these measurements. Although intuition suggests that there could be a relationship between the complexity of a software component and the number of observed failures in use, it is difficult to objectively demonstrate that this is the case. To test this hypothesis, you need failure data for a large number of components and access to the component source code for analysis. Very few companies have made a long-term commitment to collecting data about their software, so failure data for analysis is rarely available.

In the 1990s, several large companies such as Hewlett-Packard (Grady, 1993), AT&T (Barnard and Price, 1994), and Nokia (Kilpi, 2001) introduced metrics programs. They made measurements of their products and processes and used these in their quality management processes. Most of the focus was 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 metrics programs in industry in more detail.

There is little information publicly available about the current use of systematic software measurement in industry. Many companies do collect information about their software, such as the number of requirements change requests or the number of defects discovered in testing. However, it is not clear if they then use these measurements systematically to compare software products and processes or assess the impact of changes to software processes and tools. There are several reasons why this is difficult:

1. It is impossible to quantify the return on investment of introducing an organizational metrics program. There have been significant improvements in software quality over the past few years without the use of metrics so it is difficult to justify the initial costs of introducing systematic software measurement and assessment.

2. There are no standards for software metrics or standardized processes for measurement and analysis. Many companies are reluctant to introduce measurement programs until such standards and supporting tools are available.

3. In many companies, software processes are not standardized and are poorly defined and controlled. As such, there is too much process variability within the same company for measurements to be used in a meaningful way.

4. Much of the research on software measurement and metrics has focused on code-based metrics and plan-driven development processes. However, more and more software is now developed by configuring ERP systems or COTS, or by using agile methods. We don't know, therefore, if previous research is applicable to these software development techniques.

5. Introducing measurement adds additional overhead to processes. This contradicts the aims of agile methods, which recommend the elimination of process activities that are not directly related to program development. Companies that have adopted agile methods are therefore not likely to adopt a metrics program.

Software measurement and metrics are the basis of empirical software engineering (Endres and Rombach, 2003). This is a research area in which experiments on software systems and the collection of data about real projects has been used to form and validate hypotheses about software engineering methods and techniques. Researchers working in this area argue that we can only be confident of the value of software engineering methods and techniques if we can provide concrete evidence that they actually provide the benefits that their inventors suggest.

Unfortunately, even when it is possible to make objective measurements and draw conclusions from them, these will not necessarily convince decision makers. Rather, decision making is often influenced by subjective factors, such as novelty, or the extent to which techniques are of interest to practitioners. I think, therefore, that it will be many years before the results from empirical software engineering have a significant effect on software engineering practice.

24.4.1 Product metrics

Product metrics are predictor metrics that are used to measure internal attributes of a software system. Examples of product metrics include the system size, measured in lines of code, or the number of methods associated with each object class. Unfortunately, as I have explained earlier in this section, software characteristics that can be easily measured, such as size and cyclomatic complexity, do not have a clear and consistent relationship with quality attributes such as understandability and maintainability. The relationships vary depending on the development processes and technology used and the type of system that is being developed.

Product metrics fall into two classes:

1. Dynamic metrics, which are collected by measurements made of a program in execution. These metrics can be collected during system testing or after the system has gone into use. An example might be the number of bug reports or the time taken to complete a computation.

2. Static metrics, which are collected by measurements made of representations of the system, such as the design, program, or documentation. Examples of static metrics are the code size and the average length of identifiers used.

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

There is usually a clear relationship between dynamic metrics and software quality characteristics. It is fairly 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 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 15.

As I have discussed, static metrics, such as those shown in Figure 24.11, have an indirect relationship with quality attributes. A large number of different metrics have been proposed and many experiments have tried to derive and validate the relationships between these metrics and attributes such as system complexity and maintainability. None of these experiments have been conclusive but program size and control complexity seem to be the most reliable predictors of understandability, system complexity, and maintainability.

The metrics in Figure 24.11 are applicable to any program but more specific object-oriented (OO) metrics have also been proposed. Figure 24.12 summarizes

Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that 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 component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) 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 potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

Figure 24.11 Static software product metrics

Chidamber and Kemerer's suite (sometimes called the CK suite) of six object-oriented metrics (1994). Although these were originally proposed in the early 1990s, they are still the most widely used OO metrics. Some UML design tools automatically collect values for these metrics as UML diagrams are created.

El-Amam (2001), in an excellent review of object-oriented metrics, discusses the CK metrics and other OO metrics, and concludes that we do not yet have sufficient evidence to understand how these and other object-oriented metrics relate to external software qualities. This situation has not really changed since his analysis in 2001. We still don't know how to use measurements of object-oriented programs to draw reliable conclusions about their quality.

24.4.2 Software component analysis

A measurement process that may be part of a software quality assessment process is shown in Figure 24.13. Each system component can be analyzed separately using a range of metrics. The values of these metrics may then be compared for different

Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each 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 difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	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. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.
Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent, and therefore it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

Figure 24.12 The CK object-oriented metrics suite

components and, perhaps, with historical measurement data collected on previous projects. Anomalous measurements, which deviate significantly from the norm, may imply that there are problems with the quality of these components.

The key stages in this component measurement 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 that are not directly relevant to these questions need not be collected. Basili's GQM (Goal-Question-Metric) paradigm (Basili and Rombach, 1988), discussed in Chapter 26, is a good approach to use when deciding what data is to be collected.

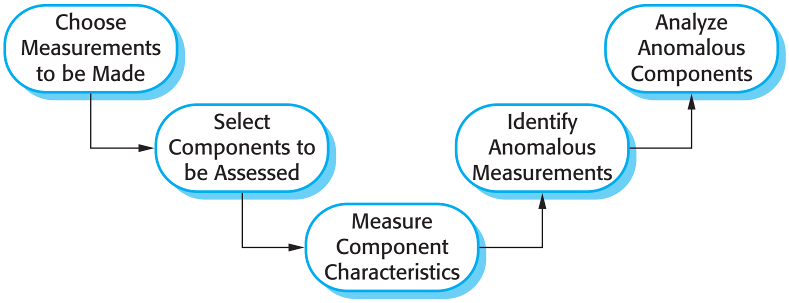
Choose Measurements to be Made
Select Components to be Assessed
Measure Component Characteristics
Identify Anomalous Measurements
Analyze Anomalous Components

Figure 24.13 The process of product measurement

2. *Select components to be assessed* You may not need to assess metric values for all of the components in a software system. Sometimes, you can select a representative selection of components for measurement, allowing you to make an overall assessment of system quality. At other times, you may wish to focus on the core components of the system that are in almost constant use. The quality of these components is more important than the quality of components that are only rarely used.

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

4. *Identify anomalous measurements* After the component measurements have been made, you then compare them with each other and to previous measurements that 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. *Analyze anomalous components* When you have identified components that have anomalous values for your chosen metrics, you should examine them to decide whether or not these 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, so may not mean that there are component quality problems.

You should always maintain collected data as an organizational resource and keep historical records of all projects even when data has not been used during a particular project. Once a sufficiently large measurement database has been established, you can then make comparisons of software quality across projects and validate the relations between internal component attributes and quality characteristics.

24.4.3 Measurement ambiguity

When you collect quantitative data about software and software processes, you have to analyze that data to understand its meaning. It is easy to misinterpret data and to make inferences that are incorrect. You cannot simply look at the data on its own—you must also consider the context where the data is collected.

To illustrate how collected data can be interpreted in different ways, consider the scenario below, which is concerned with the number of change requests made by users of a system:

> *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. She assumes that the higher the number of change requests, the less the software meets the needs of the customer.*
>
> *Handling change requests and changing the software is expensive. The organization therefore decides to modify its process with the aim of improving customer satisfaction and, at the same time, reduce the costs of making changes. The intent is that the process changes will result in better products and fewer change requests.*
>
> *Process changes are initiated to increase 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 finds it impossible to assess the effects of the process changes on the product quality.*

To understand why this kind of ambiguity can occur, you have to understand the reasons why users might make change requests:

1. The software is not good enough and does not do what customers want it to do. They therefore request changes to deliver the functionality that they require.
2. Alternatively, the software may be very good and so it is widely and heavily used. Change requests may be generated because there are many software users who creatively think of new things that could be done with the software.

Therefore, increasing the customer involvement in the process may reduce the number of change requests for products where the customers were unhappy. The process changes have been effective and have made the software more usable and suitable. Alternatively, however, the process changes may not have worked and customers may have decided to look for an alternative system. The number of change requests might decrease because the product has lost market share to a rival product and there are consequently fewer product users.

On the other hand, the process changes might lead to many new, happy customers who wish to participate in the product development process. They therefore generate more change requests. Changes to the process of handling change requests may contribute to this increase. If the company is more responsive to customers, they may generate more change requests because they know that these requests will be taken seriously. They believe that their suggestions will probably be incorporated in later versions of the software. Alternatively, the number of change requests might have increased because the beta-test sites were not typical of most usage of the program.

To analyze the change request data, you do not simply need to know the number of change requests. You need to know who made the request, how they use the software, and why the request was made. You also need information about external factors such as modifications to the change request procedure or market changes that 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 the difficulties of understanding the effects of changes and the 'scientific' approach to this problem is to reduce the number of factors that might affect the measurements made. However, processes and products that are being measured are not insulated from their environment. The business environment is constantly changing and it is impossible to avoid changes to work practice just because they may make comparisons of data invalid. As such, quantitative data about human activities cannot always be taken at face value. The reasons why a measured value changes are often ambiguous. These reasons must be investigated in detail before drawing conclusions from any measurements that have been made.

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, and so on. It includes defining standards for processes and products and establishing processes to check that these standards have been followed.
- Software standards are important for quality assurance as they represent an identification of 'best practice'. When developing software, standards provide a solid foundation for building good quality software.
- You should document a set of quality assurance procedures in an organizational quality manual. This may be based on the generic model for a quality manual suggested in the ISO 9001 standard.
- Reviews of the software process deliverables involve a team of people who check that quality standards are being followed. Reviews are the most widely used technique for assessing quality.
- In a program inspection or peer review, a small team systematically checks the code. They read the code in detail and look for possible errors and omissions. The problems detected are then discussed at a code review meeting.

- Software measurement can be used to gather quantitative data about software and the software process. You may be able to use the values of the software metrics that are collected to make inferences about product and process quality.
- Product quality metrics are particularly useful for highlighting anomalous components that may have quality problems. These components should then be analyzed in more detail.

FURTHER READING

Metrics and Models for Software Quality Engineering, 2nd edition. This is a very comprehensive discussion of software metrics covering process-, product-, and object-oriented metrics. It also includes some background on the mathematics required to develop and understand models based on software measurement. (S. H. Kan, Addison-Wesley, 2003.)

Software Quality Assurance: From Theory to Implementation. An excellent, up-to-date look at the principles and practice of software quality assurance. It includes a discussion of standards such as ISO 9001. (D. Galin, Addison-Wesley, 2004.)

'A Practical Approach for Quality-Driven Inspections'. Many articles on inspections are now rather old and do not take modern software development practice into account. This relatively recent article describes an inspection method that addresses some of the problems of using inspection and suggests how inspection can be used in a modern development environment. (C. Denger, F. Shull, *IEEE Software*, **24** (2), March–April 2007.) http://dx.doi.org/10.1109/MS.2007.31

'Misleading Metrics and Unsound Analyses'. An excellent article by leading metrics researchers that discusses the difficulties of understanding what measurements really mean. (B. Kitchenham, R. Jeffrey and C. Connaughton, *IEEE Software*, **24** (2), March–April 2007.) http://dx.doi.org/10.1109/MS.2007.49.

'The Case for Quantitative Project Management'. This is an introduction to a special section in the magazine that includes two other articles on quantitative project management. It makes an argument for further research in metrics and measurement to improve software project management. (B. Curtis et al., *IEEE Software*, **25** (3), May–June 2008.) http://dx.doi.org/10.1109/MS.2008.80.

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. Explain how standards may be used to capture organizational wisdom about effective methods of software development. Suggest four types of knowledge that might be captured in organizational standards.

24.3. Discuss the assessment of software quality according to the quality attributes shown in Figure 24.2. 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 electronically mail comments to reviewers.

24.5. Briefly describe possible standards that 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 program changes (see Chapter 26);
- The process of purchasing and installing a new computer.

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

24.7. Explain why program inspections are an effective technique for discovering errors in a program. What types of error are unlikely to be discovered through inspections?

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

24.9. Explain why it is difficult to validate the relationships between internal product attributes, such as cyclomatic complexity and external attributes, such as maintainability.

24.10. A colleague who is a very good programmer produces software with a low number of defects but consistently ignores organizational quality standards. How should her managers react to this behavior?

REFERENCES

Bamford, R. and Deibler, W. J. (eds.) (2003). 'ISO 9001:2000 for Software and Systems Providers: An Engineering Approach'. Boca Raton, Fla.: CRC Press.

Barnard, J. and Price, A. (1994). 'Managing Code Inspection Information'. *IEEE Software,* **11** (2), 59–69.

Basili, V. R. and Rombach, H. D. (1988). 'The TAME project: Towards Improvement-Oriented Software Environments'. *IEEE Trans. on Software Eng.,* **14** (6), 758–773.

Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., Macleod, G. and Merrit, M. (1978). *Characteristics of Software Quality*. Amsterdam: North-Holland.

Chidamber, S. and Kemerer, C. (1994). 'A Metrics Suite for Object-Oriented Design'. *IEEE Trans. on Software Eng.,* **20** (6), 476–93.