6

On Managing Software Quality

LEARNING OBJECTIVES

- To appreciate the need for sound measurements in determining software quality
- To critically assess various taxonomies of quality attributes
- To be able to contrast different views on software quality
- To be aware of international standards pertaining to software quality
- To know about the Capability Maturity Model
- To understand how an organization may set up its own measurement program

Software quality is an important topic. With the increasing penetration of automation in everyday life, more and more people are coming into contact with software systems, and the quality of those systems is a major concern. Quality cannot be added as an afterthought. It has to be built in from the very beginning. This chapter discusses the many dimensions of quality of both the software product and the software process.

In their landmark book *In Search of Excellence*, Peters and Waterman identify a number of key factors that set the very successful companies of the world apart from the less successful ones. One of those key factors is the commitment to quality of the very successful companies. Apparently, quality pays off.

Long-term profitability is not the only reason why attention to quality is important in software development. Because of the sheer complexity of software products and the often frequent changes that have to be incorporated during the development of software, continuous attention to, and assessment of, the quality of the product under development is needed if we ever want to realize satisfactory products. This need is aggravated by the increasing penetration of software technology into everyday life. Low-quality products will leave customers dissatisfied, will make users neglect the systems that are supposed to support their work, and may even cost lives.

One frightening example of what may happen if software contains bugs, has become known as 'Malfunction 54'. The Therac-25, a computerized radiation machine, was blamed in incidents that caused the death of two people and serious injuries to others. The deadly mystery was eventually traced back to a software bug, named 'Malfunction 54' after the message displayed at the console; see also section 1.4.2. Commitment to quality in software development not only pays off, it is a sheer necessity.

This commitment calls for careful development processes. This attention to the development process is based on the premise that the quality of a product is largely based on the quality of the process that leads to that product, and that this process can indeed be defined, managed, measured, and improved.

Besides the product--process dichotomy, a conformance--improvement dichotomy can be distinguished as well. If we impose certain quality requirements on the product or process, we may devise techniques and procedures to ensure or test that the product or process does indeed *conform to* these objectives. Alternatively, schemes may be aimed at *improving* the quality of the product or process.

Figure 6.1 gives examples of these four different approaches to quality. Most of software engineering is concerned with improving the quality of the products we develop, and the label 'best practices' in this figure refers to all of the goodies mentioned elsewhere in this book. The other three approaches are discussed in this chapter.

Before we embark on a discussion of the different approaches to quality, we will first elaborate on the notion of software quality itself, and how to measure it. When

	Conformance	Improvement
Product	ISO 9126	'best practices'
Process	ISO 9001 SQA	CMM SPICE Bootstrap

Figure 6.1 Different approaches to quality

talking about the height of people, the phrase 'Jasper is 7 ft' conveys more information than 'Jasper is tall'. Likewise, we would like to express all kinds of quality attributes in numbers. We would prefer a statement of the form 'The availability of the system is 99%' to a mere 'The availability of the system is high'. Some of the caveats of the measurement issues involved are discussed in section 6.1. In section 6.2, we will discuss various taxonomies of quality attributes, including ISO 9126. This is by no means the final word on software quality, but it is a good reference point to start from. This discussion also allows us to further illustrate some of the problems with measuring quality in quantitative terms.

'Software quality' is a rather elusive notion. Different people will have different perspectives on the quality of a software system. A system tester may view quality as 'compliance to requirements', whereas a user may view it as 'fitness for use'. Both viewpoints are valid, but they need not coincide. As a matter of fact, they probably won't. Part of the confusion about what the quality of a system entails and how it should be assessed, is caused by mixing up these different perspectives. Rather than differentiating between various perspectives on quality, Total Quality Management (TQM) advocates an eclectic view: quality is the pursuit of excellence in everything. Section 6.3 elaborates on the different perspectives on quality.

ISO, the International Standards Organization, has established several standards that pertain to the management of quality. The one most applicable to our field, the development and maintenance of software, is ISO 9001. This standard will be discussed in section 6.4.

ISO 9001 can be augmented by more specific procedures, aimed specifically at quality assurance and control for software development. The IEEE Standard for Quality Assurance Plans is meant to provide such procedures. It is discussed in section 6.5.

Software quality assurance procedures provide the means to review and audit the software development process and its products. Quality assurance by itself does not guarantee quality products. Quality assurance merely sees to it that work is done the

way it is supposed to be done.

The Capability Maturity Model (CMM)[1] is the best known attempt at directions on how to improve the development process. It uses a five-point scale to rate organizations and indicates key areas of focus in order to progress to a higher maturity level. SPICE and Bootstrap are similar approaches to process improvement. CMM is discussed in section 6.6.

Quality actions within software development organizations are aimed at finding opportunities to improve the development process. These improvements require an understanding of the development process, which can be obtained only through carefully collecting and interpreting data that pertain to quality aspects of the process and its products. Some hints on how to start such a quality improvement program are given in section 6.8.

6.1 On Measures and Numbers

> *When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science.*
> [Lord Kelvin, 1900]

> *It is the mark of an instructed mind to rest satisfied with the degree of precision which the nature of a subject admits, and not to seek exactness when only an approximation of the truth is possible.*
> [Aristotle, 330 BC]

Suppose we want to express some quality attribute, say the complexity of a program text, in a single numeric value. Larger values are meant to denote more complex programs. If such a mapping C from programs to numbers can be found, we may next compare the values of $C(P_1)$ and $C(P_2)$ to decide whether program P_1 is more complex than program P_2. Since more complex programs will be more difficult to comprehend and maintain, this type of information is very useful, e.g. for planning maintenance effort.

What then should this mapping be? Consider the program texts in figure 6.2. Most people will concur that text (a) looks less complex than text (b). Is this caused by:

- its length,
- the number of gotos,
- the number of if-statements,

[1] Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office.

(a)

```
1   procedure bubble
2       (var a: array [1..n] of integer; n: integer);
3   var i, j, temp: integer;
4   begin
5       for i:= 2 to n do
6           j:= i;
7           while j > 1 and a[j] < a[j-1] do
8               temp:= a[j];
9               a[j]:= a[j-1];
10              a[j-1]:= temp;
11              j:= j-1;
12          enddo
13      enddo
14  end;
```

(b)

```
1   procedure bubble
2       (var a: array [1..n] of integer; n: integer);
3   var i, j, temp: integer;
4   begin
5       for i:= 2 to n do
6           if a[i] ≥ a[i-1] then goto next endif;
7           j:= i;
8       loop: if j ≤ 1 then goto next endif;
9           if a[j] ≥ a[j-1] then goto next endif;
10          temp:= a[j];
11          a[j]:= a[j-1];
12          a[j-1]:= temp;
13          j:= j-1;
14          goto loop;
15      next: skip;
16      enddo
17  end;
```

Figure 6.2 Two versions of a sort routine (a) structured, (b) unstructured

– a combination of these attributes,

– something else?

Suppose we decide that the number of if-statements is what counts. The result of the mapping then is 0 for text (a) and 3 for text (b), and this agrees with our intuition. However, if we take the sum of the number of if-statements, gotos, and loops, the

result also agrees with our intuition. Which of these mappings is the one sought for? Is either of them 'valid' to begin with? What does 'valid' mean in this context?

A number of relevant aspects of measurement, such as attributes, units and scale types can be introduced and related to one another using the measurement framework depicted in figure 6.3. This framework also allows us to indicate how metrics can be used to describe and predict properties of products and processes, and how to validate these predictions.

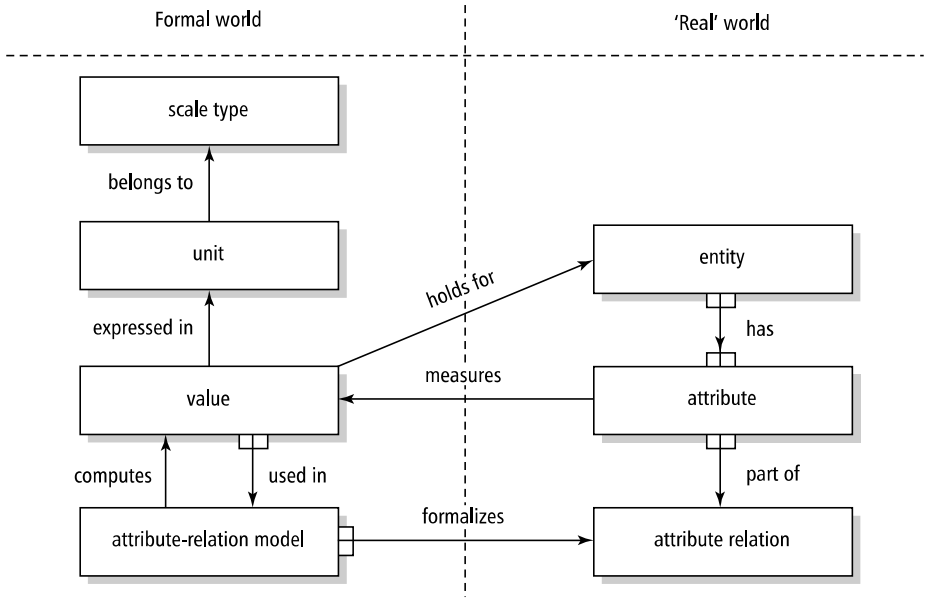

Figure 6.3 A measurement framework (*Source: B. Kitchenham, S. Lawrence Pfleeger & N. Fenton, Towards a Framework for Software Measurement Validation*, IEEE Transactions on Software Engineering **21**, *12 (1995) 1995 IEEE*)

The model in figure 6.3 has seven constituents:

- **Entity** An entity is an object in the 'real' world of which we want to know or predict certain properties. Entities need not denote material objects; projects and software are entities too.
- **Attribute** Entities have certain properties which we call attributes. Different entities may have the same attribute: both people and cars have a weight. And of course a single entity can have more than one attribute. The forks that adorn the arrow labeled 'has'

in figure 6.3 indicate that this relationship is *n-to-m*.

- **Attribute relation** Different attributes of one or more entities can be related. For example, the attributes 'length' and 'weight' of an entity 'snake' are related. Similarly, the number of man-months spent on a project is related to the cost of that project. Also, an attribute of one entity can be related to an attribute of another entity. For example, the experience of a programmer may be related to the cost of a development project he is working on.

- **Value** The former three constituents of the model reside in the 'real' world. We want to formally characterize these objects by *measuring* attributes, i.e. assigning values to them.

- **Unit** Obviously, this value is expressed in a certain unit, such as meters, seconds or lines of code.

- **Scale types** This unit in turn belongs to a certain scale type. Some common scale types are:

  - **Nominal** Attributes are merely classified: the color of my hair is gray, white or black.
  - **Ordinal** There is a (linear) ordering in the possible values of an attribute: one type of material is harder than another, one program is more complex than another.
  - **Interval** The same as ordinal, but the 'distance' between successive values of an attribute is the same, as in a calendar, or the temperature measured in degrees Fahrenheit.
  - **Ratio** The same as interval, with the additional requirement that there exists a value 0, as in the age of a software system, or the temperature measured in degrees Kelvin.
  - **Absolute** In this case we simply count the number of occurrences, as in the number of errors detected in a program.

  Note that we can sometimes measure an attribute in different units, where these units lie on different scales. For example, we can measure temperature on an ordinal scale: it either freezes, or it doesn't. We can also measure it on an interval scale: in degrees Fahrenheit or Celsius. Or we can measure it on a ratio scale: in degrees Kelvin.

- **Attribute-relation model** If there exists a relation between different attributes of, possibly different, entities in the 'real' world, we may express that relation in a formal model: the attribute-relation model. This model computes (predicts) the value of an attribute in which we are interested from the values of one or more other attributes from the model. The fork at the arrow labeled 'formalizes'

in figure 6.3 indicates that we can have more than one model for the same attribute relation.

Measurement is a mapping from the empirical, 'real' world to the formal, relational world. A **measure** is the number or symbol assigned to an attribute of an entity by this mapping. The value assigned obviously has a certain unit, e.g. lines of code. The unit in turn belongs to a certain scale, such as the ratio scale for

lines of code, or the ordinal scale for the severity of a failure.

In mathematics, the term **metric** has a very specific meaning: it describes how far apart two points are. In our field, the term is often used in a somewhat sloppy way. Sometimes it denotes a measure, sometimes the unit of a measure. We will use the term to denote the combination of:

- an attribute of an entity,
- the function which assigns a value to that attribute,
- the unit in which this value is expressed, and
- its scale type.

For each scale type, certain operations are allowed, while others are not. In particular, we can not compute the average for an ordinal scale, but only its median (middle value). Suppose we classify a system as either 'very complex', 'complex', 'average', 'simple' or 'very simple'. The assignment of numbers to these values is rather arbitrary. The only prerequisite is that a system that is classified as, say, 'very complex' is assigned a larger value than a system classified as, say, 'complex'. If we call this mapping W, the only requirement thus is: $W(\text{very complex}) > W(\text{complex}) > \ldots > W(\text{very simple})$.

Table 6.1 Example mappings for an ordinal scale

Very complex	Complex	Average	Simple	Very simple
5	4	3	2	1
100	10	5	2	1

Table 6.1 gives an example of two valid assignments of values to this attribute. Suppose we have a system with three components, which are characterized as 'very complex', 'average' and 'simple', respectively. By assigning the values from the first row, the average would be 3, so the whole system would be classified to be of average complexity. Using the values from the second row, the average would be 35, something between 'complex' and 'very complex'. The problem is caused by the fact that, with an ordinal scale, we do not know whether successive values are *equidistant*. When computing an average, we tacitly assume they are.

We often can not measure the value of an attribute *directly*. For example, the speed of a car can be determined from the values of two other attributes: a distance and the time it takes the car to travel that distance. The speed is then measured *indirectly*, by taking the quotient of two direct measures. In this case, the attribute-relation model formalizes the relation between the distance traveled, time, and speed.

We may distinguish between *internal* and *external* attributes. Internal attributes of an entity can be measured purely in terms of that entity itself. Modularity, size, defects encountered, and cost are typical examples of internal attributes. External attributes of an entity are those which can be measured only with respect to how that entity relates to its environment. Maintainability and usability are examples of external attributes. Most quality factors we discuss in this chapter are external attributes. External attributes can be measured only *indirectly*, since they involve the measurement of other attributes.

Empirical relations between objects in the real world should be preserved in the numerical relation system that we use. If we observe that car A drives faster than car B, then we would rather like our function S which maps the speed observed to some number to be such that $S(A) > S(B)$. This is called the **representation condition**. If a measure satisfies the representation condition, it is said to be a **valid measure**.

The representation condition can sometimes be checked by a careful assessment of the attribute-relation model. For example, we earlier proposed to measure the complexity of a program text by counting the number of if-statements. For this (indirect) measure to be valid we have to ascertain that:

- any two programs with the same number of if-statements are equally complex, and

- if program A has more if-statements than program B, then A is more complex than B.

Since neither of these statements is true in the real world, this complexity measure is not valid.

The validity of more complex indirect measures is usually ascertained through statistical means. Most of the cost estimation models discussed in chapter 7, for example, are validated in this way.

Finally, the scale type of indirect measures merits some attention. If different measures are combined into a new measure, the scale type of the combined measure is the 'weakest' of the scale types of its constituents. Many cost estimation formulas contain factors whose scale type is ordinal, such as the instability of the requirements or the experience of the design team. Strictly speaking, different values that result from applying such a cost estimation formula should then be interpreted as indicating that certain projects require more effort than others. The intention though is to interpret them on a ratio scale, i.e. actual effort in man-months. From a measurement-theory point of view, this is not allowed.

6.2 A Taxonomy of Quality Attributes

Some of the first elaborate studies on the notion of 'software quality' appeared in the late 1970s (McCall et al., 1977), (Boehm et al., 1978). In these studies, a number of aspects of software systems are investigated that somehow relate to the notion of software quality. In the ensuing years, a large number of people have tried to tackle this very same problem. Many taxonomies of quality factors have been published. The fundamental problems have not been solved satisfactorily, though. The various factors that relate to software quality are hard to define. It is even harder to measure them quantitatively. On the other hand, real quality can often be identified surprisingly easily.

In the *IEEE Glossary of Software Engineering Terminology*, quality is defined as 'the degree to which a system, component, or process meets customer or user needs or expectations'. Applied to software, then, quality should be measured primarily against the degree to which user requirements are met: correctness, reliability, usability, and the like. Software lasts a long time and is adapted from time to time in order to accommodate changed circumstances. It is important to the user that this is possible within reasonable costs. The customer is therefore also interested in quality factors which relate to the structure of the system rather than its use: maintainability, testability, portability, etc.

We will start our discussion of quality attributes with McCall's taxonomy. McCall distinguishes between two levels of quality attributes. Higher-level quality attributes, known as **quality factors**, are external attributes and can, therefore, be measured only indirectly. McCall introduced a second level of quality attributes, termed **quality criteria**. Quality criteria can be measured either subjectively or objectively. By combining the ratings for the individual quality criteria that affect a given quality factor, we obtain a measure for the extent to which that quality factor is being satisfied. Users and managers tend to be interested in the higher-level, external quality attributes.

For example, we can not directly measure the reliability of a software system. We may however directly measure the number of defects encountered so far. This direct measure can be used to obtain insight into the reliability of the system. This involves a theory of how the number of defects encountered relates to reliability, which can be ascertained on good grounds. For most other aspects of quality though, the relation between the attributes that can be measured directly and the external attributes we are interested in is less obvious, to say the least.

Table 6.2 lists the quality factors and their definitions, as they are used by McCall *et al.*[2] These quality factors can be broadly categorized into three classes. The first class contains those factors that pertain to the use of the software after it has become

[2]This is a rather narrow definition of software reliability. A more complete definition is contained in the IEEE Glossary of Software Engineering Terminology: 'The ability of a system or component to perform its required functions under stated conditions for a specified period of time.' It is often expressed as a probability.

Table 6.2 Quality factors (*Source: J.A. McCall, P.K. Richards & G.F. Walters*, Factors in Software Quality, *RADC-TR-77-369, US Department of Commerce, 1977.*)

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

operational. The second class pertains to the maintainability of the system. The third class contains factors that reflect the ease with which a transition to a new environment can be made. These three categories are depicted in table 6.3.

In ISO standard 9126, a similar effort has been made to define a set of quality characteristics and sub-characteristics (see table 6.4). Their definitions are given in tables 6.5 and 6.6. Whereas the quality factors and criteria as defined by McCall and others are heavily interrelated, the

ISO scheme is hierarchical: each sub-characteristic is related to exactly one characteristic.

The ISO quality characteristics strictly refer to a software *product*. Their definitions do not capture *process* quality issues. For example, security can partly be handled by provisions in the software and partly by proper procedures. Only the former is covered by the sub-characteristic 'security' of the ISO scheme. Furthermore, the sub-characteristics concern quality aspects that are *visible* to the user. Reusability, for example, is not included in the ISO scheme.

Table 6.3 Three categories of software quality factors (*Source: J.A. McCall, P.K. Richards & G.F. Walters*, Factors in Software Quality, *RADC-TR-77-369, US Department of Commerce, 1977.*)

Product operation:	
Correctness	Does it do what I want?
Reliability	Does it do it accurately all of the time?
Efficiency	Will it run on my hardware as well as it can?
Integrity	Is it secure?
Usability	Can I run it?
Product revision:	
Maintainability	Can I fix it?
Testability	Can I test it?
Flexibility	Can I change it?
Product transition:	
Portability	Will I be able to use it on another machine?
Reusability	Will I be able to reuse some of the software?
Interoperability	Will I be able to interface it with another system?

The ISO characteristics and subcharacteristics, together with an extensive set of measures, make up ISO's *external and internal quality model*. Internal quality refers to the product itself, ultimately the source code. External quality refers to the quality when the software is executed. For example, the average number of statements in a method is a measure of internal quality, while the number of defects encountered during testing is a measure of external quality.

Ultimately, the user is interested in the *quality in use*, defined in (ISO9126, 2001) as "the user's view of the quality of the software product when it is executed in a specific environment and a specific context of use." It measures the extent to which users can achieve their goals, rather than mere properties of the software (see also section 6.3). Quality in use is modeled in four characteristics: effectiveness, productivity, safety, and satisfaction. The definitions for these quality in use characteristics are given in table 6.7.

Theoretically, internal quality, external quality and quality in use are linked together: internal quality indicates external quality, which in turn indicates quality in use. In general, meeting criteria at one level is not sufficient for meeting criteria at the next level. For instance, satisfaction is partly determined by internal and external quality measures, but also includes the user's attitude towards the product. The latter has to be measured separately. Note that internal quality and external quality can be measured directly. Quality in use can in general only be measured indirectly.

Table 6.4 Quality characteristics and sub-characteristics of the external and internal quality model of ISO 9126

Characteristic	Subcharacteristics
Functionality	Suitability Accuracy Interoperability Security Functionality compliance
Reliability	Maturity Fault tolerance Recoverability Reliability compliance
Usability	Understandability Learnability Operability Attractiveness Usability compliance
Efficiency	Time behavior Resource utilization Efficiency compliance
Maintainability	Analyzability Changeability Stability Testability Maintainability compliance
Portability	Adaptability Installability Co-existence Replaceability Portability compliance

Table 6.5 Quality characteristics of the external and internal quality model of ISO 9126 (*Source:* ISO Standard 9126: *Software Quality Characteristics and Metrics. Reproduced by permission of ISO.*)

Functionality: The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.
Reliability: The capability of the software product to maintain a specified level of performance when used under specified conditions.
Usability: The capability of the software product to be understood, learned, used and be attractive to the user, when used under specified conditions.
Efficiency: The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.
Maintainability: The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.
Portability: The capability of the software product to be transferred from one environment to another.

continued on next page

Table 6.6 Quality sub-characteristics of the external and internal quality model of ISO 9126 (*Source:* ISO Standard 9126: *Software Quality Characteristics and Metrics. Reproduced by permission of ISO.*)

Suitability: The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives.
Accuracy: The capability of the software product to provide the right or agreed results or effects with the needed degree of precision.
Interoperability: The capability of the software product to interact with one or more specified systems.
Security: The capability of the software product to protect information and data so that unauthorised persons or systems cannot read or modify them and authorised persons or systems are not denied access to them.
Functionality compliance: The capability of the software product to adhere to standards, conventions or regulations in laws and similar prescriptions relating to functionality.
Maturity: The capability of the software product to avoid failure as a result of faults in the software.
Fault tolerance: The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface.
Recoverability: The capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of a failure. **Reliability compliance**: The capability of the software product to adhere to standards, conventions or regulations relating to reliability.
Understandability: The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.
Learnability: The capability of the software product to enable the user to learn its application.
Operability: The capability of the software product to enable the user to operate and control it.
Attractiveness: The capability of the software product to be attractive to the user.
Usability compliance: The capability of the software product to adhere to standards, conventions, style guides or regulations relating to usability

Time behavior: The capability of the software product to provide appropriate response and processing times and throughput rates when performing its function, under stated conditions.
Resource utilization: The capability of the software product to use appropriate amounts and types of resources when the software performs its function under stated conditions.
Efficiency compliance: The capability of the software product to adhere to standards or conventions relating to efficiency.

Analysability: The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.
Changeability: The capability of the software product to enable a specified modification to be implemented.
Stability: The capability of the software product to avoid unexpected effects from modifications of the software.
Testability: The capability of the software product to enable modified software to be validated.
Maintainability compliance: The capability of the software product to adhere to standards or conventions relating to maintainability.

Adaptability: The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered.
Installability: The capability of the software product to be installed in a specified environment.
Co-existence: The capability of the software product to co-exist with other independent software in a common environment sharing common resources.
Replaceability: The capability of the software product to be used in place of another specified software product for the same purpose in the same environment.
Portability compliance: The capability of the software product to adhere to standards or conventions relating to portability.

Quality factors are not independent. Some factors will impact one another in a positive sense, while others will do so negatively. An example from the first category is reliability versus correctness. Efficiency, on the other hand, will in general have a negative impact on most other quality factors. This means that we will have to make trade-offs between quality factors. If high requirements are decided upon for one factor, we may have to relax others. These tradeoffs are to be resolved at an early stage. An important objective of the software architecture phase is to bring these quality factors to the forefront and make the tradeoffs explicit, so that the stakeholders know what they are in for (see chapter 11). By doing so, we are better able to build in the desired qualities, as opposed to merely assess them after the fact.

Table 6.7 Quality characteristics of the quality in use model of ISO 9126 (*Source:* ISO Standard 9126: *Software Quality Characteristics and Metrics. Reproduced by permission of ISO.*)

Effectiveness: The capability of the software product to enable users to achieve specified goals with accuracy and completeness in a specified context of use.
Productivity: The capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use.
Safety: The capability of the software product to achieve acceptable levels of risk of harm to people, business, software, property or the environment in a specified context of use.
Satisfaction: The capability of the software product to satisfy users in a specified context of use.

6.3 Perspectives on Quality

> *What I (and everybody else) mean by the word quality cannot be broken down into subjects and predicates [. . .] If quality exists in an object, then you must explain why scientific instruments are unable to detect it [. . .] On the other hand, if quality is subjective, existing only [in the eye of] the observer, then this Quality is just a fancy name for whatever you'd like [. . .] Quality is not objective. It doesn't reside in the material world [. . .] Quality is not subjective. It doesn't reside merely in the mind.*
> [Robert Pirsig, *Zen and the Art of Motorcycle Maintenance*, 1974]

Users will judge the quality of a software system by the degree to which it helps them accomplish tasks and by the sheer joy they have in using it. The manager of those users is likely to judge the quality of the same system by its benefits. These benefits can be expressed in cost savings or in a better and faster service to clients.

During testing, the prevailing quality dimensions will be the number of defects found and removed, or the reliability measured, or the conformance to specifications. To the maintenance programmer, quality will be related to the system's complexity, its technical documentation, and the like.

These different viewpoints are all valid. They are also difficult to reconcile. Garvin distinguishes five definitions of software quality:

- Transcendent definition
- User-based definition
- Product-based definition
- Manufacturing-based definition

- Value-based definition.

Transcendent quality concerns innate excellence. It is the type of quality assessment we usually apply to novels. We may consider *Zen and the Art of Motorcycle Maintenance* an excellent book, we may try to give words to our admiration but these words are usually inadequate. The practiced reader gradually develops a good feeling for this type of quality. Likewise, the software engineering expert may have developed a good feeling for the transcendent qualities of software systems.

The user-based definition of quality concerns 'fitness for use' and relates to the degree in which a system addresses the user's needs. It is a subjective notion. Since different users may have different needs, they may assess a system's quality rather differently. The incidental user of a simple word-processing package may be quite happy with its functionality and possibilities while a computer scientist may be rather disappointed. The reverse situation may befall a complex system like LaTeX.

In the product-based definition, quality relates to attributes of the software. Differences in quality are caused by differences in the values of those attributes. Most of the research into software quality concerns this type of quality. It also underlies the various taxonomies of quality attributes discussed above.

The manufacturing-based definition concerns conformance to specifications. It is the type of quality definition used during system testing, whereas the user-based definition is prevalent during acceptance testing.

Finally, the value-based definition deals with costs and profits. It concerns balancing time and cost on the one hand, and profit on the other hand. We may distinguish various kinds of benefit, not all of which can be phrased easily in monetary terms:

- **Increased efficiency** Benefits are attributed to cost avoidance or reduction, and their measures are economic.

- **Increased effectiveness** This is primarily reflected through better information for decision making. It can be measured in economic terms or through key performance indicators, such as a reduced time to market.

- **Added value** Benefits enhance the strategic position of the organization, e.g. through an increased market share. The contribution of the information technology component often can not be isolated.

- **Marketable product** The system itself may be marketable, or a marketable product may be identified as a by-product of system development.

- **Corporate IT infrastructure** Communication networks, database environments and the like provide little benefit by themselves, but serve as a foundation for other systems.

Software developers tend to concentrate on the product-based and manufacturing-based definitions of quality. The resulting quality requirements can be expressed in

quantifiable terms, such as the number of defects found per man-month, or the number of decisions per module. The quality attributes discussed in the previous section fall into these categories. Such quality requirements however can not be directly mapped onto the, rather subjective, quality viewpoints of the users, such as 'fitness for use'. Nevertheless, users and software developers will have to come to an agreement on the quality requirements to be met.

One way to try to bridge this gap is to define a common language between users and software developers in which quality requirements can be expressed. This approach is taken in (Bass et al., 2003), where quality requirements are expressed in so-called quality-attribute scenarios. Figure 6.4 gives one example of how a quality attribute can be expressed in user terms. Quality attribute scenarios have a role not only in specifying requirements, but also in testing whether these requirements are (going to be) met. For example, quality attribute scenarios are heavily used in architecture assessments (see also chapter 11).

Quality attribute: Usability

Source: End user

Stimulus: Learn system features

Artifact: System

Environment: At runtime

Response: Learn tasks supplied by the system for new employees

Response measure: days on the job

Test: 90% successful completion of assigned tasks in employee test for the system, within twice the average time of an experienced user

Worst: 1 to 7 days

Plan: less than 1 day (to passing of test)

Best: less than 2 hours

Figure 6.4 A quality attribute scenario that can be used by both users and developers

Quality is not only defined at the level of the whole system. In component-based development, quality is defined at the level of a component. For services, quality is defined at the level of an individual service. The environment of the component or service, i.e. some other component or service, will require certain qualities as well. So for components and services, there is a requires and a provides aspect to quality. Since a component or service generally does not know the context in which it is going to be embedded, it is difficult to decide on the 'right' quality level. One might

then choose to offer different levels of quality. For example, a service handling video streaming may be fast with low image quality, or slow with high image quality. The user of that service then selects the appropriate quality of service (QoS) level.

Developers tend to have a mechanistic, product-oriented view on quality, whereby quality is associated with features of the product. In this view, quality is defined by looking from the program to the user (user friendliness, acceptability, etc.). To assess the quality of systems used in organizations, we have to adopt a process-oriented view on quality as well, where quality is defined by looking from the user to the program. This leads to notions like 'adequacy' and 'relevance'. For example, a helpdesk staffed with skilled people may contribute considerably to the quality of a system as perceived by its users, but this quality attribute generally does not follow from a product-based view on quality.

A very eclectic view on quality is taken in Total Quality Management (TQM). In TQM, quality applies to each and every aspect of the organization, and it is pursued by each and every employee of that organization. TQM has three cornerstones:

1. **Customer value strategy** Quality is a combination of benefits derived from a product and sacrifices required of the customer. The right balance between these benefits and sacrifices has to be sought. The key stakeholder in this balancing act is the customer, rather than the customer's boss. The attitude is not 'We know what is best for the customer', but 'Let's first determine what the customer needs'.

2. **Organizational systems** Systems encompass more than software and hardware. Other materials, humans, work practices, belong to the system as well. Moreover, systems cross unit or department boundaries. In the TQM-view, systems eliminate complexity rather than people. In TQM, culture is not dominated by power struggles. Rather, the organization takes advantage of the employees' pride in craftsmanship. Human resources are regarded as a critical resource rather than a mere cost factor.

3. **Continuous improvement** A 'traditional' environment is reactive: improvement is triggered in case of a problem or the development of a new product. In TQM, quality is pursued proactively. Errors are not viewed as personal failures which require punishment, but as opportunities for learning. Performance is not evaluated in retrospect as either good or bad, but variation in performance is analyzed statistically to understand causes of poor performance. Authority is not imposed by position and rules, but is earned by communicating a vision.

TQM thus stresses improvement rather than conformance. CMM (see section 6.6) builds on TQM, and many of the requirements engineering techniques discussed in chapter 9 owe a tribute to TQM as well.

6.4 The Quality System

ISO, the International Organization for Standardization, has developed ISO 9000, a series of standards for quality management systems. The series consists of three parts: ISO 9000:2000, ISO 9001:2000, and ISO 9004:2000. ISO 9000 gives fundamentals and vocabulary of the series of standards on quality systems. ISO 9001 integrates three earlier standards (labeled ISO 9001, ISO 9002 and ISO 9003). It specifies requirements for a quality system for any organization that needs to demonstrate its ability to deliver products that satisfy customer requirements. ISO 9004 contains guidelines for performance improvement. It is applicable after implementation of ISO 9001.

ISO 9001 is a generic standard. It can be applied to any product. A useful complement for software is ISO/IEC 90003:2004, containing guidelines for the application of ISO 9001 to computer software. It is a joint standard of ISO and IEC, the International Electrotechnical Committee. The scope of ISO/IEC 90003 is described as "This International Standard specifies requirements for a quality management system where an organization

- needs to demonstrate its ability to consistently provide a product that meets customer and applicable regulatory requirements, and
- aims to enhance customer satisfaction through the effective application of the system, including processes for continual improvement of the system and the assurance of conformity to customer and applicable regulatory requirements."

The standard is very comprehensive. It uses five perspectives from which the management of quality in software engineering is addressed:

- the systemic perspective, dealing with the establishment and documentation of the quality system itself. The quality system consists of a number of processes, such as those for software development, operation, and maintenance. These processes, and the quality system itself, have to be documented properly.
- the management perspective, dealing with the definition and management of the policies to support quality. The quality management system itself has to be developed, implemented, and regularly reviewed. This perspective describes how this is done.
- the resource perspective, dealing with the resources needed to implement and improve the quality management system, as well as to meet customer and regulatory requirements. The resources include both personnel, infrastructure, and work environment.
- the product perspective, dealing with the processes to actually create quality products, such as those pertaining to requirements engineering, design, testing, production and servicing. This perspective makes up over 60% of the standard.

- the improvement perspective, dealing with monitoring, measuring and analysis activities to maintain and improve quality.

Many organizations try, or have already tried, to obtain ISO 9000 registration. The time and cost this takes depends on how much the current process deviates from the ISO standards. If the current quality system is not already close to conforming, then ISO registration may take at least one year. ISO registration is granted when a third-party accredited body assesses the quality system and concludes that it does conform to the ISO standard. Reregistration is required every three years and surveillance audits are required every six months. ISO registration thus is a fairly drastic and costly affair, after which you certainly cannot lean back, but have to keep the organization alert.

Since software development projects have some rather peculiar characteristics (frequent changes in requirements during the development process, the rather invisible nature of the product during its development), there is a need for quality assurance procedures which are tailored towards software development. This is the topic of the next section.

6.5 Software Quality Assurance

The purpose of Software Quality Assurance (SQA) is to make sure that work gets done the way it is supposed to be done. More specifically, the goals of SQA (Humphrey, 1989) are:

- to improve software quality by appropriately monitoring the software and its development process;
- to ensure full compliance with the established standards and procedures for the software and the development process;
- to ensure that any inadequacies in the product, the process, or the standards are brought to management's attention so these inadequacies can be fixed.

Note that the SQA people themselves are not responsible for producing quality products. Their job is to review and audit, and to provide the project and management with the results of these reviews and audits.

There are potential conflicts of interest between the SQA organization and the development organization. The development organization may be facing deadlines and may want to ship a product, while the SQA people have revealed serious quality problems and wish to defer shipment. In such cases, the opinion of the SQA organization should prevail. For SQA to be effective, certain prerequisites must be fulfilled:

- It is essential that top management commitment is secured, so that suggestions made by the SQA organization can be enforced. If this is not the case, SQA

soon becomes a costly padding and a mere nuisance to the development organization;

- The SQA organization should be independent from the development organization. Its reporting line should also be independent;
- The SQA organization should be staffed with technically competent and judicious people. They need to cooperate with the development organization. If the two organizations operate as adversaries, SQA won't be effective. We must realize that, in the long run, the aims of the SQA organization and the development organization are the same: the production of high-quality products.

The review and audit activities and the standards and procedures that must be followed are described in the Software Quality Assurance Plan.

IEEE standard 730 offers a framework for the contents of a Quality Assurance Plan for software development (IEEE730, 1989). Figure 6.5 lists the entries of such a document. Appendix ?? contains a fuller description of its various constituents. IEEE standard 730 applies to the development and maintenance of critical software. For non-critical software, a subset of the requirements may be used.

IEEE standard 983 (IEEE983, 1986) is a useful complement to standard 730. IEEE Standard 983 offers further guidelines as to the contents of a quality assurance plan, the implementation of a quality assurance plan, and its evaluation and modification.

1. Purpose
2. Reference documents
3. Management
4. Documentation
5. Standards, practices, conventions, and metrics
6. Reviews and audits
7. Test
8. Problem reporting and corrective action
9. Tools, techniques, and methodologies
10. Code control
11. Media control
12. Supplier control
13. Records collection, maintenance, and retention
14. Training
15. Risk management

Figure 6.5 Main ingredients of IEEE Std 730

The Software Quality Assurance Plan describes how the quality of the software is to be assessed. Some quality factors can be determined objectively. Most factors at present can be determined only subjectively. Most often then, we will try to assess the quality by reading documents, by inspections, by walkthroughs and by peer reviews. In a number of cases, we may profitably employ tools during quality assurance, in particular, for static and dynamic analysis of program code. The actual techniques to be applied here will be discussed in the chapter on testing.

6.6 The Capability Maturity Model (CMM)

Consider the following course of events in a hypothetical software development project. Some organization is to develop a distributed library automation system. A centralized computer hosts both the software and the database. A number of local libraries are connected to the central machine through a web-based interface. The organization has some experience with library automation, albeit only with stand-alone systems.

In the course of the project, a number of problems manifest themselves. At first they seem to be disconnected and they do not alarm management. It turns out that the requirements analysis has not been all that thorough. Local requirements turn out to differ on some minor points. Though the first such deviations can be handled quite easily, keeping track of all change requests becomes a real problem after a while. When part of the system has been realized, the team starts to test the web-interface. The interface turns out to be too complex and time-consuming.

The project gets into a crisis eventually. Management has no proper means to handle the situation. It tries to cut back on both functionality and quality in a somewhat haphazard way. In the end, a rather unsatisfactory system is delivered two months late. During the subsequent maintenance phase, a number of problems are solved, but the system never becomes a real success.

Though the above description is hypothetical, it is not all that unrealistic. Many an organization has insufficient control over its software development process. If a project gets into trouble, it is usually discovered quite late and the organization has no other means but to react in a somewhat chaotic way. More often than not, speed is confused with progress.

An important step in trying to address these problems is to realize that the software development process can indeed be controlled, measured, and improved. In order to gauge the process of improving the software development process, Watts Humphrey developed a software maturity framework which has evolved into the *Capability Maturity Model* (CMM). This framework owes tribute to Total Quality Management (TQM), which in turn is based on principles of statistical quality control as formulated by Walter Shewart in the 1930s and further developed by W. Edwards Deming and Joseph Juran in the 1980s. Originally, there were separate CMM models for software engineering, systems engineering, and several others. These have now

been integrated, and carry the label CMMI[3]. The version described here is CMMI version 1.1 for software engineering and systems engineering (CMMI Product Team, 2002). CMM and CMMI were developed at the Software Engineering Institute (SEI) of Carnegie Mellon University.

In CMM (and CMMI), the software process is characterized into one of five **maturity levels**, evolutionary levels toward achieving a mature software process. To achieve a certain maturity level, a number of **process areas** must be in place. These process areas indicate important issues that have to be addressed in order to reach that level. Taken together, the process areas of a level achieve the set of goals for that level. Figure 6.6 lists the maturity levels and associated process areas of CMMI.

CMMI's maturity levels can be characterized as follows:

- **Initial** At the initial process level, the organization operates without formalized procedures, project plans, or cost estimates. Tools are not adequately integrated. Many problems are overlooked or forgotten, and maintenance poses real problems. Software development at this level can be characterized as being ad-hoc. Performance can be improved by instituting basic project management controls:
  - **Requirements management** involves establishing and maintaining an agreement with the customer on the requirements of the system. Since requirements inevitably change, controlling and documenting these requirements is important.
  - **Project planning** involves making plans for executing and managing the project. To be able to do any planning, an approved statement of work to be done is required. From this statement of work, estimates for the size of the project, resources needed, and schedule are determined, and risks to the project are identified. The results are documented in the project plan. This plan is used to manage the project; it is updated when necessary.
  - **Project monitoring and control** is concerned with the visibility of actual progress. Intermediate results have to be reviewed and tracked with respect to the project plan. When necessary, the project plan has to be realigned with reality.
  - **Supplier agreement management**. Where applicable, work done by suppliers has to be managed: plans for their part of the work have to be made, and progress of their part of the job has to be monitored.
  - **Measurement and analysis** is concerned with making sure measurements are made and their results used. First, objectives for measurement and the way measures should be collected, stored and analyzed are established. Next, the collection, storage, and analysis of data must be implemented. Finally, the results are used for decision making, and corrective actions are taken where needed.

[3] CMMI is a service mark of Carnegie Mellon University.

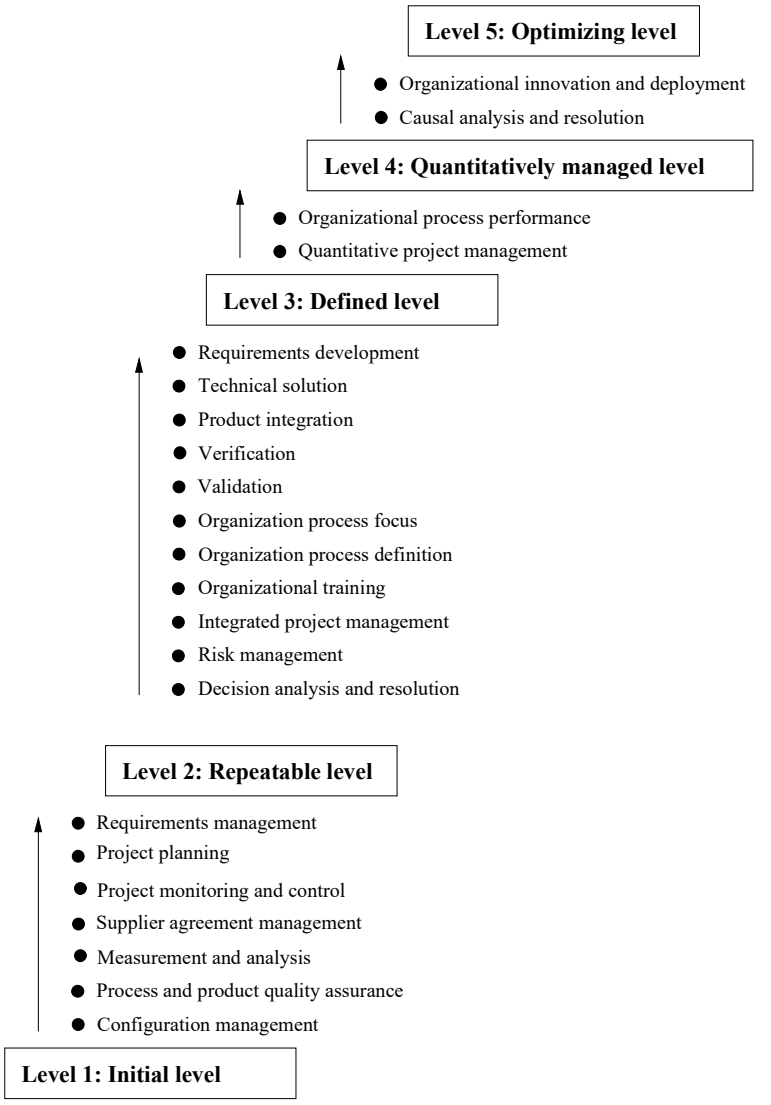

Figure 6.6 Maturity levels and associated process areas of CMMI

– **Process and product quality assurance** involves reviewing and auditing products and processes to validate that they comply with agreed upon standards and procedures.

  - **Configuration management** is concerned with establishing and maintaining the integrity of all work items during the entire project life cycle. This involves identification of configuration items and baselines, and procedures to control changes to them.

- **Repeatable** The main difference between the initial process level and the repeatable process level is that the repeatable level provides control over the way plans and commitments are established. Through prior experience in doing similar work, the organization has achieved control over costs, schedules, and change requests, and earlier successes can be repeated. The introduction of new tools, the development of a new type of product, and major organizational changes however still represent major risks at this level. The process areas needed to advance to the next level are aimed at standardizing the software process *across* the projects of the organization:

  - **Requirements development** involves the production and analysis of requirements. Requirements have to be elicited, analyzed, validated, and communicated to appropriate stakeholders. Requirements development is not a one-shot activity. Rather, requirements are identified and refined throughout the whole life cycle.

  - **Technical solution** is about design and implementation. Decisions concerning the architecture, custom development as opposed to an off the shelf solution, and modularization issues are typical ingredienst of this process area.

  - **Product integration** concerns the assembling of a complete product out of its components. This can be one stage after all components have been developed, or it can proceed incrementally. An important element of product integration is the management of interfaces, to make sure that the components properly fit together.

  - **Verification** is concerned with ensuring that the product meets its requirements. Peer reviews are an effective means for early defect detection and removal. Peer reviews, such as walkthroughs and inspections, are practices in which peers try to identify errors and areas where changes are needed.

  - **Validation** is concerned with establishing that the product fulfills its intended use. As far as possible, validation activities should be done in the intended environment in which the product is going to be used.

  - **Organization process focus** is concerned with organization process improvement. Measurements, lessons learned, project postmortems, product evaluation reports are typical sources of information to guide improvement activities. The responsibility for guiding and implementing these activities is typically assigned to a process group. In this way,

improvement of the organization's process capabilities is made a responsibility of the organization as a whole, rather than the individual project manager.

- **Organization process definition**. The organization develops and maintains a set of software process assets, such as process elements, life cycle models, guidelines for tailoring a process model. Each project uses a process built out of these process assets.
- **Organizational training**. The purpose of the training program is to develop the necessary skills and knowledge of individuals to perform their roles. Training needs are identified at the level of the organization, project and individual. The fulfillment of these needs is addressed as well.
- **Integrated project management** involves developing a project-specific software process from the organization's set of standard processes, as well as the actual management of the project using the tailored process. Since the software processes of different projects have a common ancestor, projects may now share data and lessons learned.
- **Risk management** concerns the identification of potential problems, so that timely actions can be taken to mitigate adverse effects.
- **Decision analysis and resolution** is concerned with establishing guidelines as to which issues should be subjected to formal evaluation processes, and the application of those formal processes. The selection of COTS components and architectural reviews are example arease where formal evaluation processes might be applied.

- **Defined** At the defined process level, a set of standard processes for the development and maintenance of software is in place. The organization has achieved a solid foundation, and may now start to examine their processes and decide how to improve them. Major steps to advance to the next level are:
  - **Organizational process performance**, whose purpose is to establish and maintain a quantitative understanding of the performance of the set of standard processes. Individual projects are measured, and compared against expected results as documented in a baseline. The information is not only used to assess a project, but also to quantitatively manage it.
  - **Quantitative project management**, which involves the setting of performance goals, measuring process performance, analyzing these measurements, and making the appropriate adjustments to the process in order to bring it in line with the defined limits. There is, therefore, an organization-wide measurement program and the results of it are used to continuously improve the process. An example process measure is the number of lines of code reviewed per hour.

- **Quantitatively managed** At the quantitatively managed process level, quantitative data is gathered and analyzed on a routine basis. Everything is under control, and attention may therefore shift from being reactive -- what happens to the present project? -- to being proactive -- what can we do to improve future projects? The focus shifts to opportunities for continuous improvement:
  - **Organizational innovation and deployment** is concerned with the identification and deployment of improvements. Technical improvements relate to new technologies and their orderly transition into the organization. Process improvements relate to the process in order to improve the quality of the products, the productivity of the software development organization, and reduction of the time needed to develop products.
  - **Causal analysis and resolution** is concerned with identifying common causes of defects, and preventing them from recurring.
- **Optimizing** At the final, optimizing, level, a stable base has been reached from which further improvements can be made. The step to the optimizing process level is a paradigm shift. Whereas attention at the other levels is focused on ways to improve the product, emphasis at the optimizing level has shifted from the product to the process. The data gathered can now be used to improve the software development process itself.

In 1989, Humphrey investigated the state of software engineering practice with respect to the CMM (Humphrey et al., 1989). Although this study concerned the DoD software community, there is little reason to expect that the situation was much rosier in another environment. According to his findings, software engineering practice at that time was largely at the initial level. There were a few organizations operating at the repeatable level, and a few projects operating at the defined level. No organization or project operated at the managed or optimizing levels.

In the ensuing years, a lot has happened. Many organizations have initiated a software process improvement program (SPI) to achieve a higher maturity level. Most of these improvement programs concern a move to the repeatable or defined level. The number of organizations at these levels has significantly increased since 1989. There are still few organizations or projects at the managed or optimizing level.

Reports from practical experience show that it takes about two years to move up a level. The cost ranges from $500 to $2000 per employee per year. The benefits, however, seem to easily outweigh the cost. Several companies have reported a return on investment of at least 5 to 1: every dollar invested in a process improvement program resulted in cost savings of at least $5.

To address the needs of small companies and small project teams, the Software Engineering Institute developed the Personal Software Process (PSP), a self-improvement process designed to help individuals to improve the way they work. Like the CMM, the PSP distinguishes between several maturity levels. The first step in PSP is to establish some basic measurements, such as development time and

defects found, using simple forms to collect these data. At the next level, these data are used to estimate time and quality. At still higher levels, the personal data are used to *improve* the individual's performance.

The basic principles of the CMM and the PSP are thus very similar: know thy process, measure thy performance, and base thy improvement actions on an analysis of the data gathered.

BOOTSTRAP and SPICE are two other CMM-like maturity models. BOOTSTRAP uses a separate maturity rating for each of its practices. One of the interesting features of BOOTSTRAP is that all assessment results are collected in a database, thus allowing an organization to position itself by comparing its scores with those of similar organizations.

SPICE is an international initiative and has become an international standard (ISO/IEC 15504). SPICE stands for Software Process Improvement and Capability dEtermination. SPICE distinguishes different process categories, such as the management process, customer--supplier process and engineering process. The capability (maturity) level is determined for each process category and each process. Like BOOTSTRAP, SPICE thus results in a maturity profile. The SPICE methodology places heavy emphasis on the way process assessments are performed.

6.7 Some Critical Notes

> *Software development organizations exist to develop software rather than processes.*
> (Fayad, 1997)

The massive attention of organizations to obtaining CMM or ISO 9000 certification holds the danger that focus shifts from developing software to developing processes. A certified organization, however, does not guarantee the quality of the software developed under it. A mature development process is not a silver bullet. A framed certificate definitely is not.

The SEI's Capability Maturity Model seems most appropriate for the really big companies. It is doubtful whether small companies can afford the time and money required by a process improvement program as advocated by CMM. It is also doubtful whether they can afford to implement some of the process areas, such as the 'organization process focus' process area, which requires the setting up an organization process group. Though the Personal Software Process may alleviate part of this criticism, the PSP does not have the same status as the CMM.

CMM is focused on discipline: structured work processes, strict plans, standardization. This fits bigger companies better than small ones. It also better fits activities that lend themselves to a strict approach, such as configuration management and testing. Requirements analysis and design ask for a certain amount of creativity, and a pure CMM approach may have a stifling effect here. The dichotomy noted in chapter 1 between factory-like and craft-like aspects of software engineering surfaces here as well.

CMM's original maturity levels constitute a rather crude five-point scale. If the assessment of a level 2 organization reveals that it fails the level 3 criteria on just one tiny issue, the verdict is rather harsh: the organization simply remains at level 2. This may not improve morale after two years of hard labor and significant investment. For one thing, this implication of maturity assessments places high demands on their reliability.

The rather crude assessment of organizations on a five-point scale may have other far-reaching consequences. The US government requires level 3 certification to qualify for contracts. Will this imply that level 1 and level 2 organizations are necessarily performing below standard? If level 3 certification is all that matters, is it worthwhile to aim for level 4 or 5?

CMM's original levels are like an instrument panel of an airplane with one gauge, which moreover can display only a few discrete values and thus provides the pilot with very little information. One may also envisage a software maturity 'instrument panel' with many gauges, each of which shows a lot of detail. BOOTSTRAP and SPICE are frameworks that result in a maturity profile rather than a single score. The same holds for CMMI, which comes in two variants: a **staged model** which, like the original CMM, just has five levels of maturity, and a **continuous model** in which process improvement is done on a per process area basis.

6.8 Getting Started

In the preceding sections we discussed various ways to review the quality of a software product and the associated development process. The development organization itself should actively pursue the production of quality products, by setting quality goals, assessing its own performance and taking actions to improve the development process.

This requires an understanding of possible inadequacies in the development process and possible causes thereof. Such an understanding is to be obtained through the collection of data on both the process and the resulting products, and a proper interpretation of those numbers. It is rather easy to collect massive amounts of data and apply various kinds of curve-fitting techniques to them. In order to be able to properly interpret the trends observed, they should be backed by sound hypotheses.

An, admittedly ridiculous, example is given in figure 6.7. The numbers in this table indicate that black cows produce more milk than white cows. A rather naive interpretation is that productivity can be improved significantly by repainting all the white cows.

Though the example itself is ridiculous, its counterpart in software engineering is not all that far-fetched. Many studies, for example, have tried to determine a relation between numbers indicating the complexity of software components and the quality of those components. Quite a few of those studies found a positive correlation between such complexity figures and, say, the number of defects found during testing. A straightforward interpretation of those findings then is to impose

Color	Average production
White	10
Black	40

Figure 6.7 Hypothetical relation between the color of cows and the average milk production

some upperbound on the complexity allowed for each component. However, there may be good reasons for certain components having a high complexity. For instance, (Redmond and Ah-Chuen, 1990) studied complexity metrics of a large number of modules from the MINIX operating system. Some of these, such as a module that handles escape character sequences from the keyboard, were considered justifiably complex. Experts judged a further decomposition of these modules not justified. Putting a mere upperbound on the allowed value of certain complexity metrics is too simple an approach.

An organization has to discover its opportunities for process improvements. The preferred way to do so is to follow a stepwise, evolutionary approach in which the following steps can be identified:

1. Formulate hypotheses
2. Carefully select appropriate metrics
3. Collect data
4. Interpret those data
5. Initiate actions for improvement

These steps are repeated, so that the effect of the actions is validated, and further hypotheses are formulated. By following this approach, the quest for quality will permeate your organization, which will subsequently reap the benefits.

One example of this approach is discussed in (van Genuchten, 1991). He describes an empirical study of reasons for delay in software development. The study covered six development projects from one department. Attention was focused on the collection of data relating to time and effort, viz. differences between plan and reality. A one-page data collection form was used for this purpose (see figure 6.8).

Some thirty reasons for delay were identified. These were classified into six categories after a discussion with the project leaders, and finalized after a pilot study. The reasons for delay were found to be specific to the environment.

A total of 160 activities were studied from mid 1988 to mid 1989. About 50% of the activities overran their plan by more than 10%. Comparison of planned

	Planned	Actual	Difference	Reason
Effort	---	---	---	---
Starting date	---	---	---	---
Ending date	---	---	---	---
Duration	---	---	---	---

Figure 6.8 Time sheet for each activity

and actual figures showed that the relative differences increased towards the end of the projects. It was found that one prime reason for the difference between plan and reality was 'more time spent on other work than planned'. The results were interpreted during a meeting with the project leaders and the department manager. The discussion confirmed and quantified some existing impressions. For some, the discussion provided new information. It showed that maintenance actions constantly interrupted development work. The meeting included a discussion on possible actions for improvement. It was decided to schedule maintenance as far as possible in 'maintenance weeks' and include those in quarterly plans. Another analysis study was started to gain further insights into maintenance activities.

This study provides a number of useful insights, some of which reinforce statements made earlier:

- The 'closed loop' principle states that information systems should be designed such that those who provide input to the system are also main users of its output. Application of this principle results in feedback to the supplier of data, who is thereby forced to provide accurate input. It also prevents users from asking more than they need. In the above example, the data was both collected and analyzed by the project leaders. The outcome was reported back to those same project leaders and used as a starting point for further actions.

- Local data collection should be for local use. Data collected may vary considerably between departments. Data is best used to gain insight in the performance of the department where the data is collected. Use in another department makes little sense.

- The focus should be on continuous improvement. The data collection effort was aimed at locating perceived deficiencies in the software development process. It revealed causes for these deficiencies and provided an opportunity for improvement. The question is not one of 'who is right and who is wrong', but rather 'how can we prevent this from happening again in future projects'.

- The study did not involve massive data collection. Simple data sheets were used, together with unambiguous definitions of the meaning of the various

metrics. The approach is incremental, whereby the study gives an opportunity for small improvements, and shows the way for the next study.

6.9 Summary

In this chapter, we paid ample attention to the notion of quality. Software quality does not come for free. It has to be actively pursued. The use of a well-defined model of the software development process and good analysis, design and implementation techniques are a prerequisite. However, quality must also be controlled and managed. To be able to do so, it has to be defined rigorously. This is not without problems, as we have seen in sections 6.2 and 6.3. There exist numerous taxonomies of quality attributes. For each of these attributes, we need a precise definition, together with a metric that can be used to state quality goals, and to check that these quality goals are indeed being satisfied. Most quality attributes relate to aspects that are primarily of interest to the software developers. These engineer-oriented quality views are difficult to reconcile with the user-oriented 'fitness for use' aspects.

For most quality attributes, the relation between what is actually measured (module structure, defects encountered, etc.) and the attribute we are interested in is insufficiently supported by a sound hypothesis. For example, though programs with a large number of decisions are often complex, counterexamples exist which show that the number of decisions (essentially McCabe's cyclomatic complexity) is not a good measure of program complexity. The issue of software metrics and the associated problems is further dealt with in chapter 12.

Major standards for quality systems have been defined by ISO and IEEE. These standards give detailed guidelines as regards the management of quality. Quality assurance by itself does not guarantee quality products. It has to be supplemented by a quality program within the development organization. Section 6.8 advocates an evolutionary approach to establishing a quality program. Such an approach allows us to gradually build up expertise in the use of quantitative data to find opportunities for process improvements.

We finally sketched the software maturity framework developed by the Software Engineering Institute. This framework offers a means to assess the state of software engineering practice, as well as a number of steps to improve the software development process. One of the major contributions of CMM and similar initiatives is their focus on *continuous improvement*. This line of thought has subsequently been successfully applied to other areas, resulting in, amongst others, a People-CMM, a Formal specifications-CMM, and a Measurement-CMM.

6.10 Further Reading

Fenton and Pfleeger (1996) provide a very thorough overview of the field of software metrics. The measurement framework discussed in section 6.1 is based

on (Kitchenham et al., 1995). Kaner and Bond (2004) also gives a framework for evaluating metrics. (Software, 1997b) and (JSS, 1995) are special journal issues on software metrics. Many of the articles in these issues deal with the application of metrics in quality programs.

One of the first major publications on the topic of measurement programs is (Grady and Caswell, 1987). Success factors for measurement programs can be found in (Hall and Fenton, 1997) and (Gopal et al., 2002). Pfleeger (1995) elaborates on the relation between metrics programs and maturity levels. Niessink and van Vliet (1998b) give a CMM-like framework for the measurement capability of software organizations.

The best known taxonomies of software quality attributes are given in (McCall et al., 1977) and (Boehm et al., 1978). The ISO quality attributes are described in (ISO9126, 2001) and (Côté et al., 2006). Critical discussions of these schemes are given in (Kitchenham and Pfleeger, 1996) and (Fenton and Pfleeger, 1996). Suryn et al. (2004) gives an overview of ISO/IEC 90003.

Garvin's quality definitions are given in (Garvin, 1984). Different kinds of benefit in a value-based definition of quality are discussed in (Simmons, 1996). For an elaborate discussion of Total Quality Management, see (Bounds et al., 1994) or (Ishikawa, 1985).

The Capability Maturity Model is based on the seminal work of Watts Humphrey (Humphrey, 1988, 1989). For a full description of the Capability Maturity Model Integrated, see (CMMI Product Team, 2002). Practical experiences with software process improvement programs are discussed in (Wohlwend and Rosenbaum, 1994), (Diaz and Sligo, 1997) and (Fitzgerald and O'Kane, 1999). Rainer and Hall (2003) discuss success factors, and Baddoo and Hall (2003) discuss de-motivators for SPI. A survey of benefits and costs of software process improvement programs is given in (Herbsleb et al., 1997) and (Gartner, 2001). High-maturity, CMM level 5 organizations are discussed in (Software, 2000).

The Personal Software Process (PSP) is described in (Humphrey, 1996) and (Humphrey, 1997a). BOOTSTRAP is described in (Kuvaja et al., 1994) and SPICE in (El Emam et al., 1997).

Criticisms of CMM-like approaches are found in (Fayad, 1997), (Fayad and Laitinen, 1997) and (Conradi and Fuggetta, 2002). El Emam and Madhavji (1995) discuss the reliability of process assessments.

Process improvement is the topic of several special journal issues; see (CACM, 1997), (Software, 1994). The journal *Software Process: Improvement and Practice* is wholly devoted to this topic.

Exercises

1. Define the following terms: measurement, measure, metric.
2. What is the difference between an internal and an external attribute?

3. Define the term *representation condition*. Why is it important that a measure satisfies the representation condition?
4. What is the main difference between an ordinal scale and an interval scale? And between an interval scale and a ratio scale?
5. What are the main differences between the user-based and product-based definitions of quality?
6. Which are the three categories of software quality factors distinguished by McCall?
7. Discuss the transcendent view of software quality.
8. Which of Garvin's definitions of quality is mostly used by the software developer? And which one is mostly used by the user?
9. Which quality viewpoint is stressed by ISO 9126?
10. Discuss the cornerstones of Total Quality Management.
11. What is the purpose of Software Quality Assurance?
12. Why should the Software Quality Assurance organization be independent of the development organization?
13. Why should project members get feedback on the use of quality data they submit to the Quality Assurance Group?
14. Describe the maturity levels of the Capability Maturity Model.
15. What is the major difference between level 2 and level 3 of the Capability Maturity Model?
16. What is the difference between the staged and continuous versions of CMMI?
17. Why is it important to quantify quality requirements?
18. ♠ Consider a software development project you have been involved in. How was quality handled in this project? Were quality requirements defined at an early stage? Were these requirements defined such that they could be tested at a later stage?
19. ♡ Define measurable properties of a software product that make up the quality criteria Modularity and Operability. Do these properties constitute an objective measure of these criteria? If not, in what ways is subjectivity introduced?

20. ♡ Give a possible staffing for an SQA group, both for a small development organization (less than 25 people) and a large development organization (more than 100 people).

21. ♠ Draw up a Quality Assurance Plan for a project you have been involved in.

22. ♠ One quality requirement often stated is that the system should be 'user-friendly'. Discuss possible differences between the developer's point of view and the user's point of view in defining this notion. Think of alternative ways to define system usability in measurable terms.

23. ♠ Using the classification of the Capability Maturity Model, determine the maturity level that best fits your organization. Which steps would you propose to advance the organization to a higher maturity level? Are any actions being pursued to get from the current level to a more mature one?

24. ♠ Write a critical essay on software maturity assessment, as exemplified by the Capability Maturity Model. The further reading section provides ample pointers to the literature on this topic.

25. ♡ Discuss differences between SPI approaches for large and small companies (see also (Conradi and Fuggetta, 2002)).

26. ♡ In 1988 and 1998, two surveys were conducted to assess the state of the art in software cost estimation in the Netherlands. One of the questions concerned the various stakeholders involved in developing a cost estimate. The resulting percentages were as follows:

	1988	1998
Management	48.9	75.8
Staff department	22.8	37.4
Development team	22.6	23.6
Project manager	36.7	42.3
Customer	15.4	15.9
Other	8.9	8.2
Average # of parties involved	1.55	2.03

It was concluded that the situation had improved. In 1998, the average number of parties involved had increased and this was felt to be a good sign. For each individual category, the percentage had gone up as well.

Can you think of a possibly negative conclusion from this same set of data, i.e. that the situation has become *worse* since 1988?