<http://www.dcs.qmul.ac.uk/~norman/papers/qa_metrics_article/section_4_framework.html>

<http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html>

**Rigorous Framework for Software Measurement**

The software engineering literature abounds with software ‘metrics' falling into one or more of the categories described above. So somebody new to the area seeking a small set of 'best' software metrics is bound to be confused, especially as the literature presents conflicting views of what is best practice. In the section we establish a framework relating different activities and different metrics. The framework enables readers to distinguish the applicability (and value) of many metrics. It also provides a simple set of guidelines for approaching any software measurement task.

***Measurement definitions***

Until relatively recently a common criticism of much software metrics work was its lack of rigour. In particular, much work was criticised for its failure to adhere to the basic principles of measurement that are central to the physical and social sciences. Recent work, has shown how to apply the theory of measurement to software metrics [Fenton 1991, Zuse 1991]. Central to this work is the following definition of measurement:

*Measurement* is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to characterise them according to clearly defined rules. The numerical assignment is called the *measure*.

The theory of measurement provides the rigorous framework for determining when a proposed measure really does characterise the attribute it is supposed to. The theory also provides rules for determining the scale types of measures, and hence to determine what statistical analyses are relevant and meaningful. We make a distinction between a measure (in the above definition) and a *metric*. A *metric* is a proposed measure. Only when it really does characterise the attribute in question can it truly be called a measure of that attribute. For example, the number of Lines of Code (LOC) (defined on the set of entities ‘programs’) is not a measure of ‘complexity’ or even ‘size’ of programs (although it has been proposed as such), but it is clearly a measure of the attribute of *length* of programs.

To understand better the definition of measurement in the software context we need to identify the relevant entities and the attributes of these that we are interested in characterising numerically. First we identify three classes of entities:

� **Processes**: any specific activity, set of activities, or time period within the manufacturing or development project. Relevant examples include specific activities like requirements capture, designing, coding, and verification; also specific time periods like "the first three months of project X''.

� **Products**: any artifact, deliverable or document arising out of a process. Relevant examples include source code, a design specification, a documented proof, a test plan, and a user manual.

� **Resources**: any item forming, or providing input to, a process. Relevant examples include a person or team of people, a compiler, and a software test tool.

We make a distinction between attributes of these which are internal and external:

� **Internal attributes** of a product, process, or resource are those which can be measured purely in terms of the product, process, or resource itself. For example, length is an internal attribute of any software document, while elapsed time is an internal attribute of any software process.

� **External attributes** of a product, process, or resource are those which can only be measured with respect to how the product, process, or resource relates to other entities in its environment. For example, *reliability* of a program (a product attribute) is dependent not just on the program itself, but on the compiler, machine, and user. *Productivity* is an external attribute of a resource, namely people (either as individuals or groups); it is clearly dependent on many aspects of the process and the quality of products delivered.

|  |  |  |
| --- | --- | --- |
| **ENTITIES** | **ATTRIBUTES** | |
| *Products* | *Internal* | *External* |
| Specifications | size, reuse, modularity, redundancy, functionality, syntactic correctness, ... | comprehensibility, maintainability, ... |
| Designs | size, reuse, modularity, coupling, cohesiveness, functionality, ... | quality, complexity, maintainability, ... |
| Code | size, reuse, modularity, coupling, functionality, algorithmic complexity, control-flow structuredness, ... | reliability, usability, maintainability, ... |
| Test data | size, coverage level, ... | quality, ... |
| **...** | **...** | **...** |
| *Processes* |  |  |
| Constructing specification | time, effort, number of requirements changes, ... | quality, cost, stability, ... |
| Detailed design | time, effort, number of specification faults found, ... | cost, cost-effectiveness, ... |
| Testing | time, effort, number of coding faults found, ... | cost, cost-effectiveness, stability, ... |
| **...** | **...** | **...** |
| *Resources* |  |  |
| Personnel | age, price, ... | productivity, experience, intelligence, ... |
| Teams | size, communication level, structuredness, ... | productivity, quality, ... |
| Software | price, size, ... | usability, reliability, ... |
| Hardware | price, speed, memory size, ... | reliability, ... |
| Offices | size, temperature, light, ... | comfort, quality, ... |
| **...** | **...** | **...** |

Table 4.1 provides some examples of how this framework applies to software measurement activities. Software managers and software users would most like to measure external attributes. Unfortunately, they are necessarily only measurable indirectly. For example, we already noted that productivity of personnel is most commonly measured as a ratio of: *size* of code delivered (an internal product attribute); and *effort* involved in that delivery (an internal process attribute). For the purposes of the current discussion the most important attribute that we wish to measure is 'quality' of a software system (a very high level external product attribute). It is instructive next to consider in detail the most common way of doing this since it puts into perspective much of the software metrics field.

***The defect density metric***

The most commonly used means of measuring quality of a piece of software code C is the ***defect density metric***, defined by:

http://www.dcs.qmul.ac.uk/~norman/papers/qa_metrics_article/images/p_p_r_table.gif

where size of C is normally KLOC (thousands of lines of code). Note that the external product attribute here is being measured indirectly in terms of an internal process attribute (the number of defects discovered during some testing or operational period; and an internal product attribute (size). Although it can be a useful indicator of quality when used consistently, defect density it is not actually a measure of software quality in the formal sense of the above definition of measurement. There are a number of well documented problems with this metric. In particular:

It fails to characterise much intuition about software quality and may even be more an indicator of *testing severity* than quality.

There is no consensus on what is a defect. Generally a defect can be either a *fault* discovered during review and testing (and which may *potentially* lead to an operational failure), or a *failure* that has been observed during software operation. In some studies defects means just post-release failures; in others it means all known faults; in others it is the set of faults discovered after some arbitrary fixed point in the software life-cycle (e.g. after unit testing). The terminology differs widely between organisations; fault rate, fault density and failure rate are used almost interchangeably.

It is no coincidence that the terminology defect *rate* is often used instead of defect density. Size is used only as a surrogate measure of *time* (on the basis that the latter is normally too difficult to record). For example, for operational failures defect rate should ideally be based on inter-failure times. In such a case the defect rate would be an accurate measure of *reliability*. It is reliability which we would most like to measure and predict, since this most accurately represents the user-view of quality.

There is no consensus about how to measure software size in a consistent and comparable way. Even when using the most most common size measure (LOC or KLOC) for the same programming language, deviations in counting rules can result in variations by factors of one to five.

Despite the serious problems listed above (and others that have been discussed extensively elsewhere) we accept that defect density has become the de-facto industry standard measure of software quality. Commercial organisations argue that they avoid many of the problems listed above by having formal definitions which are consistent in their own environment. In other words, it works for them, but you should not try to make comparisons outside of the source environment. This is sensible advice. Nevertheless, it is inevitable that organisations are hungry both for benchmarking data on defect densities and for predictive models of defect density. In both of these applications we do have to make cross project comparisons and inferences. It is important, therefore for broader QA issues, that we review what is known about defect density benchmarks.

Companies are (for obvious reasons) extremely reluctant to publish data about their own defect densities, even when these are relatively low. The few published references that we have found tend to be reported about anonymous third parties, and in a way that makes independent validation impossible. Nevertheless, company representatives seem happy to quote numbers at conferences and in the grey literature. Notwithstanding the difficulty of determining either the validity of the figures or exactly what was measured and how, there is some consensus on the following: in the USA and Europe the average defect density (based on number of known post-release defects) appears to be between 5 and 10 per KLOC. Japanese figures seem to be significantly lower (usually below 4 per KLOC), but this may be because only the top companies report. A well known article on 11th February 1991 in Business Week reported on results of an extensive study comparing similar 'state-of-the-art' US and Japanese software companies. The number of defects per KLOC post-delivery (first 12 months) were: 4.44 USA and 1.96 Japan. It is widely believed that a (delivered) defect density of below 2 per KLOC is good going.

In one of the more revealing of the published papers [Daskalantonakis 1992] reports that Motorola’s *six sigma quality goal* is to have ‘no more than 3.4 defects per million of output units from a project’. This translates to a an exceptionally low defect density of 0.0034 per KLOC. The paper seems to suggest that the actual defect density lay between 1 and 6 per KLOC on projects in 1990 (a figure which was decreasing sharply by 1992). Of course even the holy grail of zero-defect software may not actually mean that very high quality has been achieved. For example, [Cox 1991] reports that at Hewlett Packard, a number of systems that recorded zero post-release defects turned out to be those systems that were simply never used. A related phenomenon is the great variability of defect densities within the same system. In our own study of a major commercial system [Pfleeger et al 1994] the total 1.7 million LOC system was divided into 28 sub-systems whose median size was 70 KLOC. There was a total of 481 distinct user-reported faults for one year yielding a very low total defect density of around 0.3 per KLOC. However, 80 faults were concentrated in the subsystem which was by far the smallest (4 KLOC), and whose fault density was therefore a very high 20 per KLOC.

***Measuring size and complexity***

In all of the key examples of software measurement seen so far the notion of software ‘size’ has been a critical indirect factor. It is used as the normalising factor in the common measures of software quality (defect density) and programmer productivity. Product size as also the key parameter for models of software effort. It is not surprising therefore to note that the history of software metrics has been greatly influenced by the quest for good measures of size. The most common measure of size happens to be the simplest: Lines of Code (LOC). Other similar measures: are number of statements; number of executable statements; and delivered source instructions (DSI). In addition to the problems with these measures already discussed they all have the obvious drawback of only being defined on code. They offer no help in measuring the size of, say, a specification. Another critical problem (and the one which destroys the credibility of both the defect density metric and the productivity metric) is that they characterise only one specific view of size, namely length. Consequently there have been extensive efforts to characterise other internal product size attributes, notably *complexity* and *functionality.* In the next section we shall see how the history of software metrics has been massively influenced by this search.

[Next section - Key Metrics](http://www.dcs.qmul.ac.uk/~norman/papers/qa_metrics_article/section_5_key_metrics.html)

|  |
| --- |
| **Key Software Metrics**  ***Complexity Metrics***  Prominent in the history of software metrics has been the search for measures of complexity. This search has been inspired primarily for the reasons discussed above (as a necessary component of size) but also for separate QA purposes (the belief that only be measuring complexity can we truly understand and conquer it). Because it is a high-level notion made up of many different attributes, there can never be a single measure of software *complexity* [Fenton 1992]. Yet in the sense described above there have been hundreds of proposed complexity *metrics*. Most of these are also restricted to code. The best known are **Halstead's software science** [Halstead 1977] and **McCabe's cyclomatic number** [McCabe 1976].  http://www.dcs.qmul.ac.uk/~norman/papers/qa_metrics_article/images/halstead.gif  Figure 5.1 Halstead’s software science metrics  Halstead defined a range of metrics based on the syntactic elements in a program (the *operators* and *operands*) as shown in Figure 5.1. McCabe's metric (Figure 5.2) is derived from the program's control flowgraph, being equal to the number of linearly independent paths; in practice the metric is usually equivalent to one plus the number of decisions in the program. Despite their widespread use, the Halstead and McCabe metrics have been criticised on both empirical and theoretical grounds. Empirically it has been claimed that they are no better indicators of complexity than LOC since they are no better at predicting effort, reliability, or maintainability. Theoretically, it has been argued that the metrics are too simplistic; for example, McCabe's metric is criticised for failing to take account of data-flow complexity or the complexity of unstructured programs. This has led to numerous metrics that try to characterise different views of complexity, such as that proposed in [Oviedo 1980], that involves modelling both control flow and data flow. The approach which is more in keeping with measurement theory is to consider a range of metrics, which concentrate on very specific attributes. For example, **static path counts**, [Hatton & Hopkins, 1989] **knot** **count** [Woodward et al 1979], and **depth of nesting** [Fenton 1991].  http://www.dcs.qmul.ac.uk/~norman/papers/qa_metrics_article/images/mccabe.gif  **Figure 5.2:** Computing McCabe’s cyclomatic number  All of the metrics described in the previous paragraph are defined on individual programs. Numerous complexity metrics which are sensitive to the decomposition of a system into procedures and functions have also been proposed. The best known are those of [Henry and Kafura 1984] which are based on counts of information flow between modules. A benefit of metrics such as these is that they can be derived prior to coding, during the design stage.  ***Resource estimation models***  Most resource estimation models assume the form  effort=*f* (size).  so that size as seen as the key "cost driver". COCOMO (see figure 5.3) [Boehm 1981] is typical in this respect. In this case size is given in terms of KDSI (Thousands of Delivered Source Instructions). For reasons already discussed this is a very simplistic approach.  http://www.dcs.qmul.ac.uk/~norman/papers/qa_metrics_article/images/boehm1.gif  Figure 5.3 Simple COCOMO model  The model comes in three forms: simple, intermediate and detailed. The simple model is intended to give only an order of magnitude estimation at an early stage. However, the intermediate and detailed versions differ only in that they have an additional parameter which is a multiplicative "cost driver" determined by several system attributes.  To use the model you have to decide what type of system you are building:   * *Organic*: refers to stand-alone in-house DP systems * *Embedded*: refers to real-time systems or systems which are constrained in some way so as to complicate their development * *Semi-detached*: refers to systems which are "between organic and embedded"   The intermediate version of COCOMO is intended for use when the major system components have been identified, while the detailed version is for when individual system modules have been defined.  A basic problem with COCOMO is that in order to make a prediction of effort you have to predict size of the final system. There are many who argue that it is just as hard to predict size as it is to predict effort. Thus to solve one difficult prediction problem we are just replacing it with another difficult prediction problem. Indeed in one well known experiment managers were asked to look at complete specifications of 16 projects and estimate their implemented size in LOC. The result was an average deviation between actual and estimate of 64% of the actual size. Only 25% of estimates were within 25% of the actual.  http://www.dcs.qmul.ac.uk/~norman/papers/qa_metrics_article/images/boehm2.gif  Figure 5.4 Simple COCOMO time prediction model  While the main COCOMO model yields a prediction of total effort in person months required for project development, this output does not in itself give you a direct prediction of the project duration. However, the equations in Figure 5.4 may be used to translate your estimate of total effort into an actual schedule.  http://www.dcs.qmul.ac.uk/~norman/papers/qa_metrics_article/images/regression.gif  Figure 5.5 Regression Based Cost Modelling  Regression based cost models (see Figure 5.5) are developed by collecting data from past projects for relationships of interest (such as software size and required effort), deriving a regression equation and then (if required) incorporating additional cost drivers to explain deviations of actual costs from predicted costs. This was essentially the approach of COCOMO in its intermediate and detailed forms.  A commonly used approach is to derive a linear equation in the log-log domain that minimises the residuals between the equation and the data points for actual projects. Transforming the linear equation,  log E = log a + b\* log S  from the log-log domain to the real domain gives an exponential relationship of the form E=a\*Sb. In Figure 5.5 E is measured in person months while S is measured in KLOC.  If size were a perfect predictor of effort then every point would lie on the line of the equation, and the residual error is 0. In reality there will be significant residual error. Therefore the next step (if you wish to go that far) in regression based modelling is to identify the factors that cause variation between predicted and actual effort. For example, you might find when you investigate the data and the projects that 80% of the variation in required effort for similar sized projects is explained by the experience of the programming team. Generally you identify one or most cost drivers and assign weighting factors to model their effects. For example, assuming that medium experience is the norm then you might weight ‘low’ experience as 1.3, medium as 1.0, and high as 0.7. You use these to weight the right hand side of the effort equation. You then end up with a model of the form  Effort = (a \*Sizeb)\* F  where F is the effort adjustment factor (the product of the effort multiplier values). The intermediate and advanced versions of COCOMO contain 15 cost drivers, for which Boehm provides the relevant multiplier weights.  ***Metrics of Functionality: Albrecht's Function Points***  The COCOMO type approach to resource estimation has two major drawbacks, both concerned with its key size factor KDSI:  KDSI is not known at the time when estimations are sought, and so it also must be predicted. This means that we are replacing one difficult prediction problem (resource estimation) with another which may equally as difficult (size estimation)  KDSI is a measure of length, not size (it takes no account of functionality or complexity)  Albrecht's Function Points [Albrecht 1979] (FPs) is a popular product size metric (used extensively in the USA and Europe) that attempts to resolve these problems. FPs are supposed to reflect the user's view of a system's functionality. The major benefit of FPs over the length and complexity metrics discussed above is that they are not restricted to code. In fact they are normally computed from a detailed system specification, using the equation  FP=UFC � TCF  where UFC is the Unadjusted (or Raw) Function Count, and TCF is a Technical Complexity Factor which lies between 0.65 and 1.35. The UFC is obtained by summing weighted counts of the number of inputs, outputs, logical master files, interface files and queries visible to the system user, where:   * an *input* is a user or control data element entering an application; * an *output* is a user or control data element leaving an application; * a *logical master file* is a logical data store acted on by the application user; * an *interface file* is a file or input/output data that is used by another application; * a *query* is an input-output combination (i.e. an input that results in an immediate output).   The weights applied to simple, average and complex elements depend on the element type. Elements are assessed for complexity according to the number of data items, and master files/record types involved. The TCF is a number determined by rating the importance of 14 factors on the system in question. Organisations such as the International Function Point Users Group have been active in identifying rules for Function Point counting to ensure that counts are comparable across different organisations.  Function points are used extensively as a size metric in preference to LOC. Thus, for example, they are used to replace LOC in the equations for productivity and defect density. There are some obvious benefits: FPs are language independent and they can be computed early in a project. FPs are also being used increasingly in new software development contracts.  One of the original motivations for FPs was as the size parameter for effort prediction. Using FPs avoids the key problem identified above for COCOMO: we do not have to predict FPs; they are derived directly from the specification which is normally the document on which we wish to base our resource estimates.  The major criticism of FPs is that they are unnecessarily complex. Indeed empirical studies have suggested that the TCF adds very little in practical terms. For example, effort prediction using the unadjusted function count is often no worse than when the TCF is added [Jeffery et al 1993]. FPs are also difficult to compute and contain a large degree of subjectivity. There is also doubt they do actually measure functionality.  [Next section - measuring faults, failures and errors](http://www.dcs.qmul.ac.uk/~norman/papers/qa_metrics_article/section_6_problems.html) |

**Last modified: .**

**Recording problems (incident count metrics)**

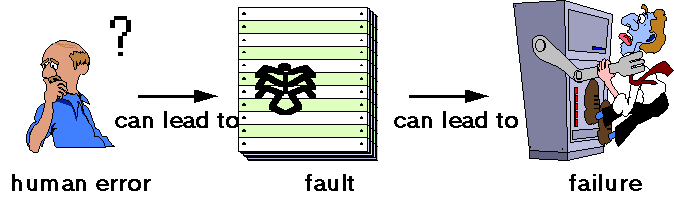
No serious attempt to use measurement for software QA would be complete without rigorous means of recording the various problems that arise during development, testing, and operation. No software developer consistently produces perfect software the first time. Thus, it is important for developers to measure those aspects of software quality that can be useful for determining

* how many problems have been found with a product
* how effective are the prevention, detection and removal processes
* when the product is ready for release to the next development stage or to the customer
* how the current version of a product compares in quality with previous or competing versions

The terminology used to support this investigation and analysis must be precise, allowing us to understand the causes as well as the effects of quality assessment and improvement efforts. In this section we describe a rigorous framework for measuring problems

***The problem with problems***

In general, we talk about problems, but Figure 6.1 depicts some of the components of a problem’s cause and symptoms, expressed in terms consistent with IEEE standard 729. [IEEE 729].



**Figure 6.1:** Software quality terminology

A **fault** occurs when a human error results in a mistake in some software product. That is, the fault is the encoding of the human error. For example, a developer might misunderstand a user interface requirement, and therefore create a design that includes the misunderstanding. The design fault can also result in incorrect code, as well as incorrect instructions in the user manual. Thus, a single error can result in one or more faults, and a fault can reside in any of the products of development.

On the other hand, a **failure** is the departure of a system from its required behavior. Failures can be discovered both before and after system delivery, as they can occur in testing as well as in operation. It is important to note that we are comparing actual system behavior with *required* behavior, rather than with *specified* behavior, because faults in the requirements documents can result in failures, too.

During both test and operation, we observe the behavior of the system. When undesirable or unexpected behavior occurs, we report it as an **incident**, rather than as a failure, until we can determine its true relationship to required behavior. For example, some reported incidents may be due not to system design or coding faults but instead to hardware failure, operator error or some other cause consistent with requirements. For this reason, our approach to data collection deals with incidents, rather than failures.

The reliability of a software system is defined in terms of incidents observed during operation, rather than in terms of faults; usually, we can infer little about reliability from fault information alone. Thus, the distinction between incidents and faults is very important. Systems containing many faults may be very reliable, because the conditions that trigger the faults may be very rare. Unfortunately, the relationship between faults and incidents is poorly understood; it is the subject of a great deal of software engineering research.

One of the problems with problems is that the terminology is not uniform. If an organization measures its software quality in terms of faults per thousand lines of code, it may be impossible to compare the result with the competition if the meaning of "fault" is not the same. The software engineering literature is rife with differing meanings for the same terms. Below are just a few examples of how researchers and practitioners differ in their usage of terminology.

To many organizations, *errors* often mean faults. There is also a separate notion of "processing error," which can be thought of as the system state that results when a fault is triggered but before a failure occurs. [Laprie 1992] This particular notion of error is highly relevant for software fault tolerance (which is concerned with how to prevent failures in the presence of processing errors).

*Anomalies* usually mean a class of faults that are unlikely to cause failures in themselves but may nevertheless eventually cause failures indirectly. In this sense, an anomaly is a deviation from the usual, but it is not necessary wrong. For example, deviations from accepted standards of good programming practice (such as use of non-meaningful names) are often regarded as anomalies.

*Defects* normally refer collectively to faults and failures. However, sometimes a defect is a particular class of fault. For example, Mellor uses "defect" to refer to faults introduced prior to coding. [Mellor 1986]

*Bugs* refer to faults occurring in the code.

*Crashes* are a special type of incident, where the system ceases to function.

Until terminology is the same, it is important to define terms clearly, so that they are understood by all who must supply, collect, analyze and use the data. Often, differences of meaning are acceptable, as long as the data can be translated from one framework to another.

We also need a good, clear way of describing what we do in reaction to problems. For example, if an investigation of an incident results in the detection of a fault, then we make a **change** to the product to remove it. A change can also be made if a fault is detected during a review or inspection process. In fact, one fault can result in multiple changes to one product (such as changing several sections of a piece of code) or multiple changes to multiple products (such as a change to requirements, design, code and test plans).

We describe the observations of development, testing, system operation and maintenance problems in terms of incidents, faults and changes. Whenever a problem is observed, we want to record its key elements, so that we can then investigate causes and cures. In particular, we want to know the following:

1. *Location*: Where did the problem occur?
2. *Timing*: When did it occur?
3. *Mode*: What was observed?
4. *Effect*: Which consequences resulted?
5. *Mechanism*: How did it occur?
6. *Cause*: Why did it occur?
7. *Severity:* How much was the user affected?
8. *Cost:* How much did it cost?

The eight attributes of a problem have been chosen to be (as far as possible) mutually independent, so that proposed measurement of one does not affect measurement of another; this characteristic of the attributes is called **orthogonality**. Orthogonality can also refer to a classification scheme within a particular category. For example, cost can be recorded as one of several pre-defined categories, such as *low* (under $100,000), *medium* (between $100,000 and $500,000) and *high* (over $500,000). However, in practice, attempts to over-simplify the set of attributes sometimes result in non-orthogonal classifications. When this happens, the integrity of the data collection and metrics program can be undermined, because the observer does not know in which category to record a given piece of information.

**Example:** Riley describes the data collection used in the analysis of the control system software for the Eurostar train (the high-speed train used to travel from Britain to France and Belgium via the Channel tunnel). [Riley 1995] In the Eurostar software problem-reporting scheme, faults are classified according to only two attributes, cause and category, as shown in Table 5.1. Note that "cause" includes notions of timing and location. For example, an error in software implementation could also be a deviation from functional specification, while an error in test procedure could also be a clerical error. Hence, Eurostar’s scheme is not orthogonal and can lead to data loss or corruption.

|  |  |
| --- | --- |
| **Cause** | **Category** |
| error in software design | category not applicable |
| error in software implementation | initialization |
| error in test procedure | logic/control structure |
| deviation from functional specification | interface (external) |
| hardware not configured as specified | interface (internal) |
| change or correction induced error | data definition |
| clerical error | data handling |
| other (specify) | computation |
|  | timing |
|  | other (specify) |

On the surface, our eight-category report template should suffice for all types of problems. However, as we shall see, the questions are answered very differently, depending on whether you are interested in faults, incidents or changes.

***Incidents***

An incident report focuses on the external problems of the system: the installation, the chain of events leading up to the incident, the effect on the user or other systems, and the cost to the user as well as the developer. Thus, a typical incident report addresses each of the eight attributes in the following way.

**Incident Report**

**Location:** such as installation where incident observed - usually a code (for example, hardware model and serial number, or site and hardware platform) that uniquely identifies the installation and platform on which the incident was observed.  
**Timing:** CPU time, clock time or some temporal measure. Timing has two, equally important aspects: real time of occurrence (measured on an interval scale), and execution time up to occurrence of incident (measured on a ratio scale).  
**Mode:** type of error message or indication of incident (see below)  
**Effect**: description of incident, such as "operating system crash," "services degraded," "loss of data," "wrong output," "no output". Effect refers to the consequence of the incident. Generally, "effect" requires a (nominal scale) classification that depends on the type of system and application.  
**Mechanism:** chain of events, including keyboard commands and state data, leading to incident. This application-dependent classification details the causal sequence leading from the activation of the source to the symptoms eventually observed. Unraveling the chain of events is part of diagnosis, so often this category is not completed at the time the incident is observed.  
**Cause**: reference to possible fault(s) leading to incident. Cause is part of the diagnosis (and as such is more important for the fault form associated with the incident). Cause involves two aspects: the type of trigger and the type of source (that is, the fault that caused the problem). The trigger can be one of several things, such as physical hardware failure; operating conditions; malicious action; user error; erroneous report while the actual source can be faults such as these: physical hardware fault; unintentional design fault; intentional design fault; usability problem.  
**Severity:** how serious the incident’s effect was for the service required from the system. Reference to a well-defined scale, such as "critical," "major," "minor". Severity may also be measured in terms of cost to the user.  
**Cost:** Cost to fix plus cost of lost potential business. This information may be part of diagnosis and therefore supplied after the incident occurs.

There are two separate notions of *mode*. On the one hand, we refer to the types of symptoms observed. Ideally, this first aspect of mode should be a measures of *what was observed* as distinct from *effect*, which is a measure of the consequences. For example, the mode of an incident may record that the screen displayed a number that was one greater than the number entered by the operator; if the larger number resulted in an item’s being listed as "unavailable" in the inventory (even though one was still left), that symptom belongs in the "effect" category.

**Example:** The IEEE standard classification for software anomalies [IEEE 1992] proposes the following classification of symptoms. The scheme can be quite useful, but it blurs the distinction between mode and effect:

* operating system crash
* program hang-up
* program crash
* input problem
* correct input not accepted
* wrong input accepted
* description incorrect or missing
* parameters incomplete or missing
* output problem
* wrong format
* incorrect result/data
* incomplete/missing
* spelling/grammar
* cosmetic
* failed required performance
* perceived total product failure
* system error message
* other
* service degraded
* loss of data
* wrong output
* no output

The second notion of mode relates to the *conditions of use* at the time of the incident. For example, this category may characterize what function the system was performing or how heavy the workload was when the incident occurred.

Only some of the eight attributes can usually be recorded at the time the incident occurs. These are:

* location
* timing
* mode
* effect
* severity

The others can be completed only after diagnosis, including root cause analysis. Thus, a data collection form for incidents should include at least these five categories.

When an incident is closed, the precipitating fault in the product has usually been identified and recorded. However, sometimes there is no associated fault. Here, great care should be exercised when closing the incident report, so that readers of the report will understand the resolution of the problem. For example, an incident caused by user error might actually be due to a usability problem, requiring no immediate software fix (but perhaps changes to the user manual, or recommendations for enhancement or upgrade). Similarly, a hardware-related incident might reveal that the system is not resilient to hardware failure, but no specific software repair is needed.

Sometimes, a problem is known but not yet fixed when another, similar incident occurs. It is tempting to include an incident category called "known software fault," but such classification is not recommended because it affects the orthogonality of the classification. In particular, it is difficult to establish the correct timing of an incident if one report reflects multiple, independent events; moreover, it is difficult to trace the sequence of events causing the incidents. However, it is perfectly acceptable to cross-reference the incidents, so the relationships among them are clear.

The need for cross-references highlights the need for forms to be stored in a way that allows pointers from one form to another. A paper system may be acceptable, as long as a numbering scheme allows clear referencing. But the storage system must also be easily changed. For example, an incident may initially be thought to have one fault as its cause, but subsequent analysis reveals otherwise. In this case, the incident’s "type" may require change, as well as the cross-reference to other incidents.

The form storage scheme must also permit searching and organizing. For example, we may need to determine the first incident due to each fault for several different samples of trial installations. Because an incident may be a first manifestation in one sample, but a repeat manifestation in another, the storage scheme must be flexible enough to handle this.

***Faults***

An incident reflects the user’s view of the system, but a fault is seen only by the developer. Thus, a fault report is organized much like an incident report but has very different answers to the same questions. It focuses on the internals of the system, looking at the particular module where the fault occurred and the cost to locate and fix it. A typical fault report interprets the eight attributes in the following way:

**Fault Report**

**Location:** within-system identifier, such as module or document name. The IEEE Standard Classification for Software Anomalies, [IEEE 1992], provides a high-level classification that can be used to report on location.  
**Timing:** phases of development during which fault was created, detected and corrected. Clearly, this part of the fault report will need revision as a causal analysis is performed. It is also useful to record the time taken to detect and correct the fault, so that product maintainability can be assessed.  
**Mode:** type of error message reported, or activity which revealed fault (such as review). The Mode classifies what is observed during diagnosis or inspection. The IEEE standard on software anomalies, [IEEE 1992], provides a useful and extensive classification that we can use for reporting the mode.  
**Effect:** failure caused by the fault. If separate failure or incident reports are maintained, then this entry should contain a cross-reference to the appropriate failure or incident reports.  
**Mechanism:** how source was created, detected, corrected. Creation explains the type of activity that was being carried out when the fault was created (for example, specification, coding, design, maintenance). Detection classifies the means by which the fault was found (for example, inspection, unit testing, system testing, integration testing), and correction refers to the steps taken to remove the fault or prevent the fault from causing failures.  
**Cause:** type of human error that led to fault. Although difficult to determine in practice, the cause may be described using a classification suggested by Collofello and Balcom: [Collofello and Balcom 1985]: a) communication: imperfect transfer of information; b) conceptual: misunderstanding; or c) clerical: typographical or editing errors  
**Severity:** refer to severity of resulting or potential failure. That is, severity examines whether the fault can actually be evidenced as a failure, and the degree to which that failure would affect the user  
**Cost:** time or effort to locate and correct; can include analysis of cost had fault been identified during an earlier activity

***Changes***

Once a failure is experienced and its cause determined, the problem is fixed through one or more changes. These changes may include modifications to any or all of the development products, including the specification, design, code, test plans, test data and documentation. Change reports are used to record the changes and track the products most affected by them. For this reason, change reports are very useful for evaluating the most fault-prone modules, as well as other development products with unusual numbers of defects. A typical change report may look like this:

**Change Report**

**Location:** identifier of document or module affected by a given change. **Timing:** when change was made  
**Mode:** type of change  
**Effect:** success of change, as evidenced by regression or other testing  
**Mechanism:** how and by whom change was performed  
**Cause:** corrective, adaptive, preventive or perfective  
**Severity:** impact on rest of system, sometimes as indicated by an ordinal scale  
**Cost:** time and effort for change implementation and test

[Next section - Standards for measurement: CMM, ISO and GQM](http://www.dcs.qmul.ac.uk/~norman/papers/qa_metrics_article/section_7_standards.html)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Measurement Frameworks and Standards**  ***Goal-Question-Metrics (GQM)***  Many software metrics programmes have failed because they had poorly defined, or even non-existent objectives. To counter this problem Vic Basili and his colleagues at Maryland University developed a rigorous goal oriented approach to measurement [Basili and Rombach 1988]. Because of its intuitive nature the approach has gained widespread appeal. The fundamental idea is a simple one; managers proceed according to the following three stages:   1. Set goals specific to needs in terms of *purpose*, *perspective* and *environment*. 2. Refine the goals into quantifiable questions that are tractable. 3. Deduce the metrics and data to be collected (and the means for collecting them) to answer the questions.   Figure 7.1 illustrates how several metrics might be generated from a single goal.  http://www.dcs.qmul.ac.uk/~norman/papers/qa_metrics_article/images/gqm.gif  **Figure 7.1:** Example of Deriving Metrics from Goals and Questions  The figure shows that the overall goal is to evaluate the effectiveness of using a coding standard. To decide if the standard is effective, several key questions must be asked. First, it is important to know who is using the standard, so that you can compare the productivity of the coders who use the standard with the productivity of those who do not. Likewise, you probably want to compare the quality of the code produced with the standard with the quality of non-standard code. To address these issues, it is important to ask questions about productivity and quality.  Once these questions are identified, you must analyze each question to determine what must be measured in order to answer the question. For example, to understand who is using the standard, it is necessary to know what proportion of coders is using the standard. However, it is also important to have an experience profile of the coders, explaining how long they have worked with the standard, the environment, the language, and other factors that will help to evaluate the effectiveness of the standard. The productivity question requires a definition of productivity, which is usually some measure of effort divided by some measure of product size. As shown in the figure, the metric can be in terms of lines of code, function points, or any other metric that will be useful to you. Similarly, quality may be measured in terms of the number of errors found in the code, plus any other quality measures that you would like to use.  In this way, you generate only those measures that are related to the goal. Notice that, in many cases, several measurements may be needed to answer a single question. Likewise, a single measurement may apply to more than one question. The goal provides the purpose for collecting the data, and the questions tell you and your project how to use the data.  **Example:** AT&T used GQM to help determine which metrics were appropriate for assessing their inspection process. [Barnard and Price 1994] Their goals, with the questions and metrics derived, are shown in Table 7.1.  **Table 7.1:** Examples of AT&T goals, questions and metrics   |  |  |  | | --- | --- | --- | | ***Goal*** | ***Questions*** | ***Metrics*** | | Plan | How much does the inspection process cost? | Average effort per KLOC Percentage of reinspections | |  | How much calendar time does the inspection process take? | Average effort per KLOC Total KLOC inspected | | Monitor and control | What is the quality of the inspected software? | Average faults detected per KLOC Average inspection rate Average preparation rate | |  | To what degree did the staff conform to the procedures? | Average inspection rate Average preparation rate Average lines of code inspected Percentage of reinspections | |  | What is the status of the inspection process? | Total KLOC inspected | | Improve | How effective is the inspection process? | Defect removal efficiency Average faults detected per KLOC Average inspection rate Average preparation rate Average lines of code inspected | |  | What is the productivity of the inspection process? | Average effort per fault detected Average inspection rate Average preparation rate Average lines of code inspected |   GQM is in fact only one of a number of approaches for defining measurable goals that have appeared in the literature: the other most well known approaches are:   * **Quality Function Deployment Approach (QFD)** is a technique that evolved from Total Quality Management principles that aims at deriving indicators from the user's point of view The QFD method uses simple matrices (the so-called 'House of Quality') with values weighted according to the judgement of the customer; * **Software Quality Metrics (SQM)** as exemplified by [McCall et al 1977] was developed to allow the customer to assess the product being developed by a contractor. In this case a set of quality factors is defined on the final product; the factors are refined into a set of criteria, which are further refined into a set of metrics (as shown in Figure 3.2). Essentially this is a model for defining external product quality attributes in terms of internal attributes.   ***Process Improvement and the Capability Maturity Model (CMM)***  Process improvement is an umbrella term for a growing movement underpinned by the notion that all issues of software quality revolve around improving the software development process. Central to this movement has been the work of the Software Engineering Institute (SEI) at Carnegie Mellon promoting the **Capability Maturity Model (CMM)**. The CMM has its origins in [Humphrey 1989] and the latest version is described in [Paulk et al 1994]. The development of the CMM was commissioned by the US DOD as a ramification of the problems experienced in their software procurement. They wanted a means of assessing the suitability of potential contractors. The CMM is a five-level model of a software development organisation's process maturity (based very much on TQM concepts), as shown in Figure 1.  http://www.dcs.qmul.ac.uk/~norman/papers/qa_metrics_article/images/cmm.gif  Figure 1: CMM  By means of an extensive questionnaire, follow-up interviews and collection of evidence, software organisations can be 'graded' into one of the five maturity levels, based primarily on the rigour of their development processes. Except for level 1, each level is characterised by a set of *Key Process Areas* (KPA's). For example, the KPA's for level 2 are: requirements management, project planning, project tracking, subcontract management, quality assurance and configuration management. The KPA's for level 5 are defect prevention, technology change management, and process change management.  Ideally, companies are supposed to be at level 3 at least to be able to win contracts from the DOD. This important commercial motivation is the reason why the CMM has such a high profile. Few companies have managed to reach as high as level 3; most are at level 1. Only very recently has there been evidence of any level 5 organisations; the best known is the part of IBM responsible for the software for NASA's space shuttle programme [Keller 1992].  The CMM is having a huge international impact, and this impact has resulted in significantly increased awareness and use of software metrics. The reason for this is that metrics are relevant in KPAs throughout the model. Table 7.2 presents an overview of the types of measurement suggested by each maturity level, where the selection depends on the amount of information visible and available at a given maturity level. Level 1 measurements provide a baseline for comparison as you seek to improve your processes and products. Level 2 measurements focus on project management, while level 3 measures the intermediate and final products produced during development. The measurements at level 4 capture characteristics of the development process itself to allow control of the individual activities of the process. A level 5 process is mature enough and managed carefully enough to allow measurements to provide feedback for dynamically changing the process during a particular project’s development.   |  |  |  | | --- | --- | --- | | ***Maturity Level*** | ***Characteristics*** | ***Type of Metrics to Use*** | | 5. Optimizing | Improvement fed back to the process | Process plus feedback for changing the process | | 4. Managed | Measured process | Process plus feedback for control | | 3. Defined | Process defined and institutionalized | Product | | 2. Repeatable | Process dependent on individuals | Project management | | 1. Initial | Ad hoc | Baseline |   Despite its international acceptance, the CMM is not without criticism. The most serious accusation concerns the validity of the five-level scale itself. There is, as yet, no convincing evidence that higher rated companies produce better quality software. There have also been concerns regarding the questionnaire [Bollinger and McGowan 1991]. A European project (funded under the ESPRIT programme) that is closely related to the CMM is **Bootstrap** [Woda and Schynoll 1992]. The Bootstrap method is also a framework for assessing software process maturity; the key differences are that individual projects (rather than just entire organisations) can be assessed and the results of assessments can be *any* real numbers between 1 and 5. Thus, for example, a department could be rated at 2.6, indicating that it is 'better' than level 2 maturity (in CMM) but not good enough for level 3 in CMM.  The most recent development in the process improvement arena is SPICE (Software Process Improvement and Capability dEtermination). This is an international project [ISO/IEC 1993] whose aim is to develop a standard for software process assessment, building on the best features of the CMM, Bootstrap, and ISO9003 (described below).  ***Relevant Standards***  There are now literally hundreds of national and international standards which are directly or indirectly concerned with software quality assurance. A general criticism of these standards is that they are overtly subjective in nature and that they concentrate almost exclusively on the development processes rather than the products [Fenton et al 1993]. Despite these criticisms the following small number of generic software QA standards are having a significant impact on software metrics activities for QA.  *ISO9000 series and TickIT*  In Europe and also increasingly in Japan, the pre-eminent quality standard to which people aspire is based around the international standard, ISO 9001[ISO 9001]. This general manufacturing standard specifies a set of 20 requirements for a quality management system, covering policy, organisation, responsibilities, and reviews, in addition to the controls that need to be applied to life cycle activities in order to achieve quality products. ISO 9001 is not specific to any market sector; the software 'version' of the standard is ISO 9003 [ISO 9003]. The ISO 9003 standard is also the basis of the **TickIT** initiative that was sponsored by the UK Department of Trade and Industry [TickIT 1992]. Companies apply to become TickIT-certified (most of the key IT companies have already successfully achieved this certification); they must be fully re-assessed every three years.  Different countries have their own national standards based on the ISO9000 series. For example, in the UK, the equivalent is the BS5750 series. The EEC equivalent to ISO 9001 is EN29001.  *ISO 9126 Software product evaluation: Quality characteristics and guidelines for their use*  This is the first international standard to attempt to define a framework for evaluating software quality [ISO9126, Azuma 1993]. The standard defines software quality as:  '*The totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs*'.  Heavily influenced by the SQM approach described above, ISO 9216 asserts that software quality may be evaluated by six characteristics: functionality, reliability, efficiency, usability, maintainability and portability. Each of these characteristics is defined as a 'set of attributes that bear' on the relevant aspect of software, and can be refined through multiple levels of subcharacteristics. Thus, for example, *reliability* is defined as  '*A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time*.'  while *portability* is defined as  '*A set of attributes that bear on the capability of software to be transferred from one environment to another*.'  Examples of possible definitions of subcharacteristics at the first level are given, but are relegated to Annex A, which is not part of the International Standard. Attributes at the second level of refinement are left completely undefined. Some people have argued that, since the characteristics and subcharacteristics are not properly defined, ISO 9126 does not provide a conceptual framework within which comparable measurements may be made by different parties with different views of software quality, e.g., users, vendors and regulatory agencies. The definitions of attributes like reliability also differ from other well-established standards. Nevertheless, ISO9126 is an important milestone in the development of software quality measurement.  *IEEE 1061: Software Quality Metrics Methodology*  This standard [IEEE 1061] was finalised in 1992. It does not prescribe any product metrics, although there is an Appendix which describes the SQM approach. Rather it provides a methodology for establishing quality requirements and identifying, analysing, and validating software quality metrics |