# Software Engineering Measures

How good is a program? How reliable will a software system be once it is installed? How much more testing should I do? How many more bugs can I expect to find? How much will the testing cost? How difficult will it be to maintain a system? How much will it cost to build a new system similar to one we built five years ago? How long will it take?

Software engineers face questions like these every day of their professional lives, and they are difficult questions to answer. Some of them address attributes of software systems that, conceivably, can be measured directly. Others ask for predictions, and we usually try to answer these based on trends, or patterns of measurements over a period of time. In either case, the ability to make appropriate measurements is a fundamental skill for software engineers.

## 1. Software Engineering Measures

We are all familiar with the common measures of properties in the physical world: length, height, distance, weight, speed, acceleration, time, brightness, loudness, electrical current, etc. These and other measures have been used by scientists and engineers for hundreds of years. It is not so obvious, however, what properties of software systems can and should be measured.

We can characterize some software measures as *static*, meaning that they can be derived from examination of the software itself (usually in the form of source or object code, or perhaps in terms of a design document). Other measures can be characterized as *dynamic*, meaning that they can only be derived from observation of the execution of the software.

We can also characterize some measures as being *basic* or *directly measurable* quantities, and others as *composite, derived,* or *indirectly measurable* quantities. A common example of a derived measure is *productivity*, which we define informally as the amount of something produced in a unit of time. Usually, the amount produced is directly measurable, as is the time it takes. The mathematical operation of dividing the amount measure by the time measure gives the productivity measure.

---

This document is taken from the SEI educational materials package "Lecture Notes on Engineering Measurement for Software Engineers" by Gary Ford, document number CMU/SEI-93-EM-9, copyright 1993 by Carnegie Mellon University. Permission is granted to make and distribute copies for noncommercial purposes.

Computer scientists and software engineers have done a lot of research trying to define the important measures of software engineering. One of the most significant efforts was undertaken over the last four years at the Software Engineering Institute (SEI), a federally funded research and development center at Carnegie Mellon University. Researchers at the SEI, assisted by more than 60 specialists from industry, academia, and government, identified four direct measures and several indirect measures that software engineering organizations can use to improve their software development processes. The properties or attributes of software that are directly measurable are *size*, *effort*, *schedule*, and *quality*. The results of the SEI research on measures of these properties are elaborated in Sections 2-4.

Another property whose measure is widely regarded as fundamentally important is *performance*, which can be defined in several ways. Clearly, a performance measure is a dynamic software measure. There are a few other software properties that are generally believed to be important but which we don't yet know how to measure very well. Among these are *reliability* and *complexity*. Measures of all these attributes are discussed in Sections 5-7.

Finally, there are other attributes of software that seem important but that we don't know how to measure at all. These include *maintainability*, *usability,* and *portability*. Measurement of these attributes is discussed in Section 8.

Because software engineering is such a new discipline, we are struggling not only with the question of *what* to measure but also with *how* to measure. Measurement in the physical world has evolved to a state where there are well-defined standard units of measurement for almost everything. There are also standard ways of computing the composite measures that are accepted within specific branches of science, engineering, manufacturing, and management.

This is not yet the case in software engineering, although recent work has begun to suggest measurement standards. We will look at the issue of standard ways of measuring in our discussions of the different kinds of measures.

We expect that computer scientists will continue to make progress in finding ways to measure software, and that software engineers will continue to make progress in finding ways to measure the software engineering process, both in terms of basic measures and derived measures. For a student of software engineering, our best advice is to learn now to make the measurements that we do know how to make and to watch for new measurements to be developed over the coming years.

All the measures we present in the subsequent sections can and should be studied in more detail by students of software engineering. Our immediate goal is to introduce the measures and encourage you to begin using them, even in a limited way, to gain insight into your own individual and class programming projects. Being able to use these measures will be important when you become a professional software engineer.

## 2. Program Size Measures

Perhaps the most obvious and most fundamental measure of software is *program size*. Many questions that software engineers must answer related to costs, schedules, progress, reuse, and productivity are in some way based on the size of the software product being built.

The most widely used size measure is a count of *source lines of code* (SLOC). Unfortunately, there are as many definitions of what to count as there are people doing the counting. Some people count executable statements but not comments; some include declarations while others exclude them; some count physical statements and others count logical statements. Published information on software measures that depend on this measure is therefore difficult to interpret and compare.

One SEI report says this about measurement of source code size: "Historically, the primary problem with measures of source code size has not been in coming up with numbers—anyone can do that. Rather, it has been in identifying and communicating the attributes that describe exactly what those numbers represent."

What is needed is a way of adding precision to software size measurements. Remember that precision is an indication of the repeatability of a measurement. If several people are asked to measure the size of a program, we would like them all to measure the same things and get the same answer.

The results of an experiment illustrate this point. The C program shown in Figure 1 was given to about 80 people, and they were asked how many source lines of code are in the program. Figure 2 shows how many votes each of the possible answers received.

```
#define  LOWER 0     /* lower limit of table */
#define  UPPER 300   /* upper limit */
#define  STEP  20    /* step size */

main()   /* print a Fahrenheit-Celsius conversion table */
{
  int fahr;
  for(fahr=LOWER; fahr<=UPPER; fahr=fahr+STEP)
    printf("%4d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Figure 1. C program used in the measurement experiment

The precision of a measurement of source lines of code does not depend on the numbers used in counting (everyone agrees to use the nonnegative integers), so it must depend on what we choose to count. A comprehensive definition of what kinds of statements or constructs in a program to count is necessary before precise measurement is possible.

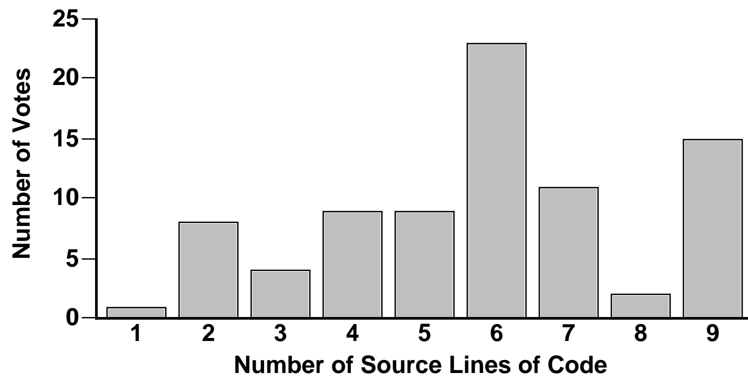


Figure 2. Results of the experiment

**Class Exercise**

A fragment of a Pascal implementation of a binary tree search algorithm is shown below. Count the number of physical lines of code and the number of logical lines of code. Collect these counts from all class members and then plot the results as two histograms (as in Figure 2).

```
repeat
  if tree = nil
    then
      finished := true
    else
      with tree^ do
        if key < data
          then
            tree := left
          else if key > data
            then
              tree := right
            else
              finished = true
until finished;
```

## 2.1. What Can We Count?

The SEI research on software measures led to the creation of checklists for software engineers to use in defining software measures. The checklists provide a way of defining exactly what is to be counted and reported. An excerpt of the checklist for source statement counts is shown in Figure 3. Your instructor has a copy of the complete checklist.

The first thing to notice in this excerpt is the measurement unit, which is either physical source lines or logical source statements. The choice is indicated in the box at the top. Then to define a particular size measure, we simply check in the appropriate column whether to include or exclude a particular statement type from the count. The

## Definition Checklist for Source Statement Counts

Definition name: ***Physical Source Lines of Code*** Date: 8/7/92

Originator: SEI

| **Measurement unit:** | | |
|---|---|---|
| | **Physical source lines** | ✔ |
| | **Logical source lines** | |

| **Statement type** **Definition** ✔ **Data array** ☐ | Order | **Includes** | **Excludes** |
|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | |
| 1 Executable **Order of precedence ->** | 1 | ✔ | |
| 2 Nonexecutable | | | |
| 3 Declarations | 2 | ✔ | |
| 4 Compiler directives | 3 | ✔ | |
| 5 Comments | | | |
| 6 On their own lines | 4 | | ✔ |
| 7 On lines with source code | 5 | | ✔ |
| 8 Banners and nonblank spacers | 6 | | ✔ |
| 9 Blank (empty) comments | 7 | | ✔ |
| 10 Blank lines | 8 | | ✔ |
| 11 | | | |
| 12 | | | |

| **How produced** **Definition** ☐ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 Programmed | ✔ | |
| 2 Generated with source code generators | ✔ | |
| 3 Converted with automated translators | ✔ | |
| 4 Copied or reused without change | ✔ | |
| 5 Modified | ✔ | |
| 6 Removed | | ✔ |
| 7 | | |
| 8 | | |

Figure 3. Portion of the SEI definition checklist for source statement counts

example in Figure 3 indicates that we should count executable statements, declarations, and compiler directives, but not comments.

Notice also that some programming languages allow more than one kind of statement on a line. If we are counting physical source lines, then whether or not to count a line may be ambiguous when there are two or more kinds of statements on that line. The checklist solves this problem by allowing us to specify a precedence of statement types (the boxes just to the left of the "Includes" column). When a line contains two or more statement kinds, we count it if the highest precedence statement is specified as included.

An important observation is that different kinds of measures are needed for different purposes. For example, when planning project costs, a software engineer may want to treat new code differently from reused code—new code probably costs a lot more to develop. Similarly, code written for the developers' own use and never delivered to the

| Attribute class | Describes and distinguishes |
|---|---|
| **statement type** | executable and nonexecutable statements, and if a statement is a declaration, a comment, a compiler directive, or a blank line |
| **how produced** | code programmed by a software engineer, created by source code generation tools, converted from another program or language by an automated translator, copied or reused unchanged from another program, modified from a previous version of the same program, or removed from a previous version |
| **origin** | new work (no prior existence) and prior work; source of prior work, such as a previous version of the program, a different program, a commercial program library, a reuse library, etc. |
| **usage** | code that is part of the primary product, or is external to or in support of the primary product |
| **delivery** | code is to be delivered (either as source code or object code), or it is only used internally |
| **functionality** | operative or inoperative (meaning unused, unreferenced, or inaccessible code) |
| **replications** | code that may be replicated in the final delivered product, such as code that is copied, expanded, or instantiated during compilation and linking, or code that is replicated during installation (such as in a distributed system) |
| **development status** | where the code is in the development process: planned, designed, coded, unit tests completed, integrated into components, etc. |
| **language** | programming language used; for example, a software engineer may want to distinguish code in assembly language from code in a high-level language |

Figure 4. Source code attribute classes

customer, such as a little program that generates test data, may need to be counted separately from the delivered product.

To account for all the different uses of software size measures, nine different, independent classes of attributes of statements in programs have been identified. For each size measure, we must specify whether or not to count source lines in those attribute classes. Figure 4 contains brief definitions of the classes.

For different programming languages, there may be special cases to be considered in defining how to count statements. The complete checklist includes space to list clarifications to the general rules of what to include or exclude in a measure of source lines of code.

To help ensure precise measurement, the SEI checklist has been designed so that when source statements are counted, each statement or physical line has exactly one value per attribute. For this to happen, values within an attribute must be both mutually exclusive and collectively exhaustive.

## 2.2. What Should We Count?

Although the use of the statement count checklist allows us to be very precise in what we measure, there are literally thousands of different ways of filling out the checklist to define a particular measure. Which should we use?

The SEI has recommended that the simplest measure, physical source lines, is perhaps the best measure at this time. In the future, some of the more complex measures may prove useful; but right now, we simply do not know how to get more value from the complex measures than from the simplest one.

Counting physical lines is almost as easy as counting carriage return characters in the source code. It is also almost independent of the programming language. Automated tools that count physical lines are usually much simpler than tools that count logical statements. Furthermore, if a software organization has a relatively uniform programming style and commenting style (sometimes enforced by coding standards), there may be a strong relationship between the physical line count and the logical statement count.

**Discussion Question 1**

As an alternative to the simple process of counting carriage returns, some organizations suggest the equally simple process of counting semicolons (in languages like Pascal, Ada, and C). Discuss the adequacy of such a measure, using the Pascal code fragment in the class exercise above (page 4) as an example.

**Class Exercise**

We have seen that it is easier to measure physical lines of code than logical lines of code in a program. If there is a strong mathematical relationship between the two measures, then we can make the easy measurement and use it to get a fairly good estimate of the other measure.

To test this hypothesis, first use the size definition checklist to define physical lines of code and logical lines of code. Then each member of the class should make the measurements for a few of his or her own programs. Plot the relationship between the two measures. Is it linear? If you are familiar with curve-fitting techniques, use them to establish a mathematical relationship between the two measures.

# 3. Effort and Schedule Measures

The next fundamental software measures we want to examine are *effort* and *schedule*. Reliable measures for effort are prerequisites for reliable measures of software cost. The principal means we have for managing and controlling costs and schedules is through planning and tracking the human resources we assign to individual tasks and activities.

Good measurement of effort, combined with good measurement of software size, can give us a variety of measures of *productivity*, which in oversimplified terms is the amount of product divided by the amount of effort. Effort and size measurements, when collected on several projects over a period of time, provide the data needed to calibrate a *cost estimation model*. Such a model is another valuable tool for software engineers, one that you will examine in detail later in your studies.

### 3.1. Effort Measures

Some of the most common units of measurement of effort are *labor-month* (sometimes still called *man-month*), *staff-week*, and *staff-hour*. The SEI recommends the last of these as the best unit, citing two main reasons. First, the length of a month or a week is not well defined because of different company practices, vacations, overtime, and other factors. Second, because a goal of effort measurement is to help organizations improve their software development process, tracking individual activities at the week or month level does not give detailed enough information about the process.

Precise measurement of effort is facilitated by using a checklist to define exactly what kinds of effort should and should not be counted. A portion of the SEI checklist is shown in Figure 6. Your instructor has a copy of the complete checklist.

The effort checklist, like the size checklist, is structured in sections defined by different classes of attributes. Those classes are defined in Figure 5.

| **Attribute class** | **Describes and distinguishes** |
|---|---|
| type of labor | direct and indirect labor: labor costs that can be charged directly to the project or contract, and those that cannot |
| hour information | regular or overtime work, and salaried or hourly workers |
| employment class | regular company employees, whether full-time or part-time, and employees brought in to work on a specific project task, such as consultants and subcontractors |
| labor class | workers by the types of work they do: managers at various levels, analysts, designers, programmers, documentation specialists, support staff, etc. |
| activity | software development activities and maintenance activities |
| product-level functions | functions of software development, such as design, coding, testing, and documentation; organized by major functional element, by customer release, and by system |

Figure 5. Effort attribute classes

The checklist serves two separate but related purposes. First, by placing check marks in the columns titled “Totals include” and “Totals exclude,” we can produce a *definition* of what is being counted. Second, by placing check marks in the column titled “Report totals,” we can define the content of a specific *effort report*. Different reports will be needed for different purposes: we may want separate reports for effort expended on

**Staff-Hour Definition Checklist**

| Definition Name: | Total System Staff-Hours for Development | Date: | 7/28/92 |
|---|---|---|---|
| | | Originator: | |
| | | Page: | 1 of 3 |

| Type of Labor | Totals include | Totals exclude | Report totals |
|---|---|---|---|
| Direct | ✔ | | |
| Indirect | | ✔ | |
| **Hour Information** | | | |
| Regular time | | | ✔ |
| Salaried | ✔ | | |
| Hourly | ✔ | | |
| Overtime | | | ✔ |
| Salaried | | | |
| Compensated (paid) | ✔ | | |
| Uncompensated (unpaid) | ✔ | | |
| Hourly | | | |
| Compensated (paid) | ✔ | | |
| Uncompensated (unpaid) | ✔ | | |

| Product-Level Functions continued | Totals include | Totals exclude | Report totals |
|---|---|---|---|
| **System-Level Functions** | | | ✔ |
| (Software effort only) | | | |
| System requirements & design | | | |
| System requirements analysis | ✔ | | |
| System design | ✔ | | |
| Software requirements analysis | ✔ | | |
| Integration, test, & evaluation | | | |
| System integration & testing | ✔ | | |
| Testing & evaluation | ✔ | | |
| Production and deployment | | ✔ | |
| Management | ✔ | | |
| Software quality assurance | ✔ | | |
| Configuration management | ✔ | | |
| Data | ✔ | | |
| Training | | | |
| Training of development employees | ✔ | | |
| Customer training | | ✔ | |
| Support | ✔ | | |

Figure 6. Portion of staff-hour definition checklist

design or on testing, or we may want a report on overtime needed or on the effort of contract employees or consultants.

**Discussion Question 2**

Look carefully at the complete SEI effort reporting checklist (available from your instructor). How many of the different activity attributes and product-level function attributes do you recognize as applicable to your own class programming work? How would you measure your own work in each of the applicable categories?

**Class Project**

Use the checklist to define precisely the effort measures to be made and reported for a large class programming project. Choose one class member to be the *project administrator*, who is responsible for organizing and reporting the measures. Design a schedule and a reporting system through which each class member makes and reports his or her own personal effort measures.

At the end of the project, determine project costs associated with major functions such as requirements analysis and specification, design, coding, and testing. Use a typical figure of $50 per hour to determine the total value of your product to your customer.

## 3.2. Schedule Measures

Although software engineers are routinely asked to provide information to be used in the creation and tracking of project schedules, most scheduling tasks are the responsibility of engineering project managers. For that reason, we will present only a brief discussion of schedule measures.

The SEI recommends that software project managers adopt structured methods for defining two aspects of schedules and reports: the calendar *dates* (both planned and actual) associated with project milestones, reviews, audits, and deliverables; and the *exit* or *completion criteria* associated with each date.

An all-too-common error in designing schedules is to choose the wrong kinds of milestones for a project, especially those for which the completion criteria are difficult or impossible to identify. For example, even if we have a good estimate of the size of a piece of software under development, we should avoid choosing a milestone such as "code 50% written" to be met halfway through the allotted time for the project. It is unlikely that we will be able to recognize when the code is half written; and, if we can, it is not uncommon to discard some of that code later because of unforeseen problems. Also, we have all experienced the phenomenon of the last 10% of the code requiring as much time to complete as the first 90%.

Examples of good completion criteria for software project milestones include:

- internal review held
- formal review with customer held
- all action items closed
- document entered under configuration management
- system delivered to customer
- customer comments received
- changes incorporated
- customer sign-off obtained

## 4. Quality Measures

For many engineered products, the quality of the product depends to a great extent on the quality of the raw materials and the quality of the machines used in the manufacture of the product. This is not the case with software. In fact, recent research suggests that the most important factor in achieving quality in a software product is the quality of the software process used to create that product.

There are many aspects of the software process that an organization might want to improve in order to produce better products. Underlying any process improvement effort is the idea that we must be able to recognize improvement when it happens, and this requires measurement.

Many modern definitions of quality are based on two fundamental ideas: *freedom from defects* and *suitability for use*. These ideas suggest the most basic quality measures we should adopt: counts of *defects* and *problem reports*. If used carefully and repeatedly, these measures will exhibit trends that provide insight into a wide range of opportunities for software process improvement. They are also among the very few direct measures we have for software quality, and they are the basis for quantifying other software quality attributes such as reliability, correctness, completeness, efficiency, and usability.

To facilitate precise reporting of defects and problems, the SEI has produced a problem count definition checklist. A portion of this checklist is shown in Figure 8. Your instructor has a copy of the complete checklist.

The problem count definition checklist, like the size checklist, is structured in sections defined by different classes of attributes, as defined in Figure 7.

| **Attribute class** | **Describes and distinguishes** |
|---|---|
| **problem status** | points in the problem analysis and correction process: open or closed; recognized, evaluated, or resolved |
| **problem type** | software defect or other kind of problem (hardware, operating system, user mistake, operations mistake, new requirement, enhancement); for a software defect, whether it is a defect in requirements, design, code, operational document, test case, etc. |
| **uniqueness** | new and unique defect, or a duplicate of another reported defect |
| **criticality** | degree of disruption to a user when the problem is encountered |
| **urgency** | degree of importance given to the evaluation, resolution, and closure of the problem |
| **finding activity** | the activity that uncovered the problem, such as synthesis, inspection, formal review, testing, customer use |
| **finding mode** | operational or non-operational environment where defect was found |

Figure 7. Defect attribute classes

## Problem Count Definition Checklist

**Software Product ID [Example V1 R1]** **Page 1**
**Definition Identifier: [ Problem Count A]** **Definition Date [ 01/02/92 ]**

| **Attributes/Values** | **Definition [ ]** | | **Specification [ X ]** | |
|---|---|---|---|---|
| **Problem Status** | Include | Exclude | Value Count | Array Count |
| Open | ✔ | | ✔ | |
| Recognized | | | | |
| Evaluated | | | | ✔ |
| Resolved | | | | ✔ |
| Closed | ✔ | | ✔ | |
| **Problem Type** | Include | Exclude | Value Count | Array Count |
| Software defect | | | | |
| Requirements defect | ✔ | | ✔ | |
| Design defect | ✔ | | ✔ | ✔ |
| Code defect | ✔ | | ✔ | ✔ |
| Operational document defect | ✔ | | ✔ | ✔ |
| Test case defect | | ✔ | | |
| Other work product defect | | ✔ | | |
| | | | | |
| Other problems | | | | |
| Hardware problem | | ✔ | | |
| Operating system problem | | ✔ | | |
| User mistake | | ✔ | | |
| Operations mistake | | ✔ | | |
| New requirement/enhancement | | ✔ | | |
| | | | | |
| Undetermined | | | | |
| Not repeatable/Cause unknown | | ✔ | | |
| Value not identified | | ✔ | | |
| **Uniqueness** | Include | Exclude | Value Count | Array Count |
| Original | ✔ | | | |
| Duplicate | | ✔ | ✔ | |
| Value not identified | | ✔ | | |

Figure 8. Portion of the problem count definition checklist

As we have seen before, this checklist has “Include” and “Exclude” columns to specify precisely what characteristics of defects and problems are to be counted. The result of counting problems according to the attributes included and excluded is a *single number*, the total number of problems.

The column titled “Value Count” is used to specify other values that are to be reported. For example, in Figure 8, check marks in this column appear in the rows for open and closed problems; requirements, design, code, and operational document defects; and duplicate problems. Each of these values should be reported separately, in addition to the total number of problems. Notice that this means that duplicate problems are not included in the total, but they are reported separately.

The last column is titled “Array Count.” Check marks in this column identify multi-dimensional arrays of counts that should be reported. For example, in Figure 8 there

are check marks in this column for two attributes in the problem status class (evaluated and resolved), and check marks for three attributes in the problem type class (design, code, operational document). This means that we want a report that contains two columns and three rows, with cells containing the count of problems with each of the six possible pairs of attributes (evaluated design defects, resolved design defects, ...).

Use of this checklist allows an organization to define precisely what defects and problems are to be counted and reported. This data can then be used to identify trends and to help improve the organization's software process.

**Discussion Question 3**

You have probably used a variety of commercial software packages such as word processors, spreadsheets, drawing programs, or games. You have also probably encountered a situation where the behavior of the program was not what you expected. In such situations, how can you determine whether the problem is a user mistake, an error in the user manual, or an actual error in the program? How much does the answer to the previous question matter to the user? To the software engineers who must resolve the problem?

Have you ever heard a programmer say, "That's not a bug, it's an undocumented feature!"

## 5. Performance Measures

Performance is an important attribute of many engineered products or systems. Two familiar examples are the performance of our cars ("0 to 60 in 8.9 seconds") and our computers ("2.5 million instructions per second"). These illustrate the two most common kinds of performance measures: *response time*, or how long it takes to accomplish a particular task, and *throughput*, or how many tasks can be completed in a unit of time.

**Discussion Question 4**

What are some other everyday examples of performance measures? What kinds of performance measures might be important to the designers and users of a long-distance telephone system, an airliner, an automatic banking machine, a washing machine, a water heater, or the food preparation equipment at a fast-food restaurant? Are these measures of response time, throughput, or something else?

The ability to quantify and to measure software performance is an important tool for software engineers. First, it gives us a way to state system requirements more precisely. For example, we can require that a compiler be able to compile 2,000 lines per minute, a word processor be able to scroll a full page of text in one second or check the spelling of 500 words per second, or a computer game be able to move the animated figures as fast as the refresh rate of the display screen (typically 1/60 second). Without

the ability to measure performance, we might be tempted to accept requirements such as "the system must be efficient" or "the system must be as fast as possible."

Second, if we can measure software performance, we can demonstrate that our system satisfies quantitative performance requirements. Suppose you are developing a system for a customer, and your contract says you don't get paid until you demonstrate that the system complies with the requirements specification. Would you rather try to demonstrate that it can compile 2,000 lines per minute or that it is as fast as possible?

There are two basic performance measurement techniques. The first, called *event recording*, identifies events that are important to system performance and then records when they happen. For example, suppose we want to measure the time between a user's keystroke and the appearance of the corresponding character on the screen. (Note that this is a response time measurement.) To make this measurement, we can add some event-recording code to our system. At the point where the keystroke is first recognized (often via an interrupt handler), we record the time from the internal system clock. At the point where the routine that actually modifies the pixels of the screen display completes the drawing of the character, we get another reading from the system clock. The difference between the two times is recorded in a database. A series of recorded events gives us the data to determine minimum, maximum, and average response times.

This approach to event recording is commonly known as *instrumenting the code*. The parallel with other engineering measurement is clear: mechanical engineers often use mechanical instruments to measure mechanical systems; electrical engineers use electronic instruments to measure electronic systems; software engineers use software instruments to measure software systems.

The second performance measurement technique is called *monitoring*. It is usually a sampling technique: at regular intervals we record appropriate data on the state of the system. For example, we may be interested in identifying performance bottlenecks in a system—is too much time spent waiting for input and output requests to complete, or in computation, or in allocating and freeing dynamic storage, or somewhere else? Rather than trying to instrument the whole system and record every event, we can incorporate a sampling monitor that records, at regular intervals, the value of the computer's program counter. We choose the interval to be long enough that the monitoring does not interfere with the running of the system, but short enough to be sure that the important routines cannot run to completion between sampling measurements. Analysis of the resulting data can give a good picture of where the system spends its time.

Another kind of monitoring, *hardware monitoring*, is often used in real-time control systems. These systems commonly receive signals from various *sensors* and then send signals to *effectors* that perform an action. For example, in a *fly-by-wire* aircraft control system, movement of the control stick or yoke by the pilot results in a signal to the control system (hardware and software). The software must interpret that signal and send an appropriate signal to the aircraft flight control surfaces (such as the ailerons and elevator) within a specified amount of time (usually measured in milliseconds).

Because the input and output signals exist outside the controlling computer, an electronic probe can be attached to the wires carrying those signals. The response time (the time between input signal and output signal) can be measured directly with an appropriate instrument, such as an oscilloscope.

Software performance measurements usually should be viewed as a kind of experiment, rather than as an absolute measure. This is because the performance of a system almost always depends on the inputs to the system at the time the measurement is made. In fact, we often qualify a measurement with phrases like *average*, *peak load*, or *worst case* to indicate the conditions under which the experiment was conducted.

A particularly important kind of measurement experiment is called a *benchmark*. It is conducted using a specific, carefully chosen set of inputs. Those inputs must be representative of the inputs that the system will receive in normal use, and they must be reproducible. This allows us to conduct the experiment many times to determine performance differences resulting from design or implementation changes in the system.

One of the most common uses of benchmarks is to compare performance of different models of computer systems. Using one or more carefully designed and reproducible computational tasks, we can conduct experiments and make quantitative statements about the relative performance of the different systems.

**Discussion Question 5**

What kind of measurement technique could be used to demonstrate that a word processor can check the spelling of 500 words per second? What other response time and throughput measures might be appropriate for word processors?

**Discussion Question 6**

In retail stores, cash registers have given way to *point-of-sale terminals* that are connected to one or more computer systems. Many of these terminals have the capability to read the magnetic encoding strip on credit cards, contact the credit card company, and get purchase authorization with just a single keystroke. What kinds of performance requirements might you expect if you were asked to design the software system that performs purchase authorization? Which are response time requirements and which are throughput requirements?

## 6. Reliability Measures

The reliability of a system, software or other, is often extremely important to the user of that system. The lack of reliability cannot be tolerated in a safety-critical software system, such as the control system for a nuclear power plant or the flight control software of a modern airliner, where a failure can result in loss of life. Thus it is important

for a software engineer to be able to state quantitative reliability requirements and to demonstrate that those requirements have been satisfied.

Software reliability cannot be measured directly. It is generally inferred or computed from other measures of the behavior of the software. Some ideas from the other engineering disciplines suggest how we might measure it.

Many physical systems, such as machines, are subject to “wear and tear” or physical degradation of moving parts. After a period of time, a part may be unable to perform its function and need to be replaced. A common measure of how long the system can operate between failure of a critical part is “mean time between failures” (MTBF), which is the average of the times between successive failures. Notice that this is a statistical measure rather than a direct measure. We sometimes can compute the MTBF by observing the system over a long period of time. For some systems, we can also do component testing to determine the MTBF for each component and then use statistical techniques to predict the MTBF of the entire system.

If we have good MTBF data, we can design a preventive maintenance schedule that anticipates failures and replace the parts before the failure occurs.

A similar measure is “mean time to repair” (MTTR), which is the average amount of time it takes to repair the machine after a failure. Many machines are designed with parts or modules that can be quickly exchanged; the design goal is to minimize MTTR. Not surprisingly, such a design goal has application in software engineering.

Engineers also compute the *availability* of a system, which is the percentage of time that the system is ready for use. It can be computed from MTBF and MTTR as:

$$availability = \left(1 - \frac{\text{MTTR}}{\text{MTTR} + \text{MTBF}}\right) \times 100$$

**Discussion Question 7**

Issues of reliability and availability sometimes strike very close to home when the system involved is our car. Which components on a car seem to have a low MTBF? High MTBF? Of these, which have high and low MTTR? What parts or components of a car are usually involved in preventive maintenance? Are these the same as the ones you identified as having a low MTBF?

Software does not have parts that wear out, and it is not usually the case that the user of the software can pull out a faulty module and slide in a good one. Still, the basic ideas of reliability and availability are important to software engineers and software users.

We generally define software reliability as the probability that the software will perform its task under stated conditions for a stated period of time. When the software does fail,

it is not because a part has worn out, but because it has encountered a set of conditions or input values that it cannot handle correctly. So to determine the reliability, we need to know the probability that the user will give the system inputs that cause failure. This means we need to gather statistics on the *use* of the software as well as the software itself. Statistics on the patterns of use of the software is called an *operational profile*.

Statistics of this nature are used in a variety of ways by software engineers. In requirements analysis, you may be able to identify potential requirements that, in fact, will never be needed by the user. During testing, in order to reach a desired degree of reliability, you can devote a lot of time to testing features that will be heavily used and little time to testing features that are rarely used.

The ability to measure or predict reliability serves three general purposes in software engineering. First, it allows us to understand and make tradeoffs between reliability and other software characteristics, such as performance, cost, and schedule. Second, it can allow us to track progress during software testing; this is useful both for recognizing when we have done enough testing and for predicting when the testing will be completed. Third, as with many software measures, it allows us to determine the effect of using new tools or methods to develop software.

Detailed discussion of software reliability requires a good knowledge of statistics and mathematics, so we will not try to cover it here. It is an increasingly important topic, and we recommend that it be included in the education of all software engineers.

**Discussion Question 8**

Computer scientists have expended much effort in pursuit of program *correctness,* which we define informally as the equivalence (in some mathematical sense) of the requirements specification and the code. You may have studied the various methods that have been developed to do proofs of correctness.

Software engineers might suggest, “Correctness is a red herring; it is unachievable and unnecessary. Reliability is much more important.”

Consider a software package that you use frequently, such as a word processor or compiler. Suppose you have experienced 100% reliability of the software under the conditions of your use, although there are known defects in parts of the software you never use. Technically, the software is incorrect, but to you it is perfectly satisfactory. Which is more important? Which costs more to achieve?

Suggest arguments on both sides of this issue. You may want to distinguish correctness at the module level from correctness at the system level. Consider also the question of whether a requirements specification can be shown to be correct.

Do you detect a fundamental difference between the philosophies of computer science and software engineering in this discussion?

## 7. Complexity Measures

Two important facts of software engineering are that software systems evolve over time and that making that evolution happen requires expenditure of resources. To minimize the costs, we would like to have software that can easily evolve in desirable ways. Computer scientists and software engineers have been working for years to figure out how to make this happen.

One step in the process was the recognition that some programs are easier to change ("maintain") than others. If there were some way to measure *maintainability*, these programs would get a high measurement. However, we could not find a direct measure of maintainability.

The next step was the recognition that these maintainable programs tended to be easy to understand. This quality made it simpler for maintenance programmers to design and implement changes. If there were some way to measure *understandability*, these programs would get a high measurement. However, we could not find a direct measure of understandability.

Then it was suggested that understandability seemed to be related to an abstract quality called *complexity*, which may be a structural attribute and might be measurable from source code. Although the connections from complexity to understandability to maintainability to lower cost are somewhat tenuous, researchers have been inventing complexity measures with great energy. To date, more than 100 such measures have appeared in the computer science technical literature.

Complexity measurement is an interesting example when we consider the questions of what *can* be measured and what *should* be measured. Many of the suggested complexity measures can be measured quite easily by running the source code through a measurement tool. Many of them are inherently interesting, especially to scientists, who are usually interested in discovering new facts. Unfortunately, they are much less interesting to engineers, who want to build better products and reduce costs.

The ideal situation might be to have complexity (or understandability or maintainability) measures that could be applied very early in the development process, so that the resulting system could be maintained economically. So far, we do not know how to use any of the 100 complexity measures to do this.

If we can't *guarantee* low maintenance costs, the second best situation would be to be able to *predict* what the maintenance costs will actually be. Having confidence in our maintenance cost estimates allows us to make competent decisions about when and whether to revise a software system. That is much better than making the commitment to revise a product and then discovering that it costs ten times as much as we thought.

Currently, we don't yet know which of the 100 complexity measures is a good predictor of maintenance costs. We hope that ongoing research will change this situation.

## 8. Other Software Measures

Software engineers have identified a number of other properties or qualities or attributes of software that seem to be desirable but that we currently have no way of measuring. Figure 9 lists some of these. Because of many of their names, these properties are often referred to as the *ilities* (pronounced like "ill at ease," which describes our emotional state when asked to measure them).

| | |
|---|---|
| Accessibility | The extent to which software facilitates selective use or maintenance of its components |
| Adaptability | The ease with which software allows differing system constraints and user needs to be satisfied |
| Compatibility | The ability of two or more systems to exchange information |
| Fault tolerance | The built-in capability of a system to provide continued correct execution in the presence of a limited number of hardware or software faults |
| Integrity | The extent to which unauthorized access to or modification of software or data can be controlled in a computer system |
| Interoperability | The ability of two or more systems to exchange information and to mutually use the information that has been exchanged |
| Maintainability | The ease with which software can be maintained |
| Portability | The ease with which software can be transferred from one computer system or environment to another |
| Reusability | The extent to which a module can be used in multiple applications |
| Robustness | The extent to which software can continue to operate correctly despite the introduction of invalid inputs |
| Testability | The extent to which software facilitates both the establishment of test criteria and the evaluation of the software with respect to those criteria |

Figure 9. Some unmeasurable attributes of software

**Discussion Question 9**

Although we cannot measure the *ilities* directly, we may have strong intuition that certain measurable attributes are closely related to one of them. For example, we may design software so that all the system-dependent information is encapsulated in a single module. To port the software to a different computer system might then require recoding of only that module. We could argue that, intuitively, the number of modules that use system-dependent information is a measure of portability.

Suggest other measures that you believe intuitively are related to the unmeasurable *ilities*.