

AD-A266 959



Educational Materials
CMU/SEI-93-EM-9

2



Carnegie-Mellon University
Software Engineering Institute

Lecture Notes on Engineering Measurement for Software Engineers

Gary Ford

April 1993

DTIC
ELECTE
JUL 20 1993
S A D

This document has been approved
for public release and sale; its
distribution is unlimited.

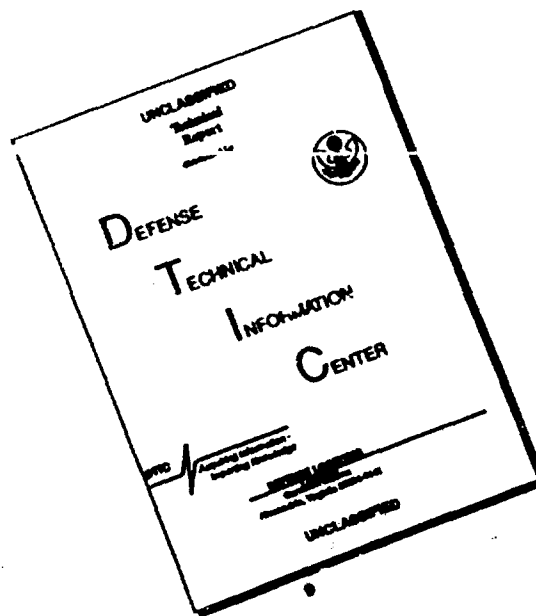
416208

93-16291



93 7 19 678

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST
QUALITY AVAILABLE. THE COPY
FURNISHED TO DTIC CONTAINED
A SIGNIFICANT NUMBER OF
PAGES WHICH DO NOT
REPRODUCE LEGIBLY.

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs on the basis of race, color, national origin, sex or handicap. Violation of Title VI of the Civil Rights Act of 1964, Title IX of the Education Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state or local laws, or code of conduct.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religious creed, ancestry, birthplace, marital status, sexual orientation, or violation of federal, state or local laws, or codes of conduct. While the federal government does not prohibit employers from discriminating on the basis of sexual orientation, EEOC guidance issued in July 1992 encourages employers to avoid discrimination on the basis of sexual orientation.

For more information regarding this statement, it is referred to the following Carnegie Mellon University, 4800 Forbes Avenue, Pittsburgh, Pa. 15260, telephone (412) 268-6400 or the Vice President for Enrollment, Carnegie Mellon University, 4800 Forbes Avenue, Pittsburgh, Pa. 15260, telephone (412) 268-2000.

Lecture Notes on Engineering Measurement for Software Engineers



Gary Ford

Academic Education Project

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

DTIC QUALITY INSPECTED 8

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

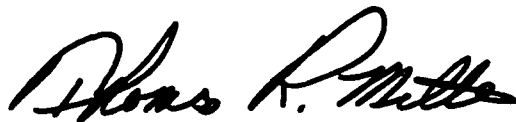
SEI Joint Program Office
ESC/ENS
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

The Software Engineering Institute is sponsored by the U.S. Department of Defense.

This report was funded by the U.S. Department of Defense.

Copyright © 1993 by Carnegie Mellon University.

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Service. For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

Copies of this document are also available from Research Access, Inc., 3400 Forbes Avenue, Suite 302, Pittsburgh, PA 15213.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

Preface	v
Information for Instructors	1
1. Objectives	1
2. Where in the Curriculum to Use the Materials	2
3. Pedagogical Considerations	2
4. Suggestions for Class Exercises and Projects	3
5. Suggested Answers to Discussion Questions	5
5.1. Questions from "Introduction to Engineering Measurement"	5
5.2. Questions from "Measurement Theory for Software Engineers"	12
5.3. Questions from "Software Engineering Measures"	13
6. Further Reading	16

Lecture Notes

Introduction to Engineering Measurement
Measurement Theory for Software Engineers
Software Engineering Measures

Classroom Materials

Transparency Masters
Software Measure Forms for Duplication

Lecture Notes on Engineering Measurement for Software Engineers

Abstract: Measurement is a fundamental skill for engineers. To facilitate teaching software engineering measurement, materials are provided to support three lectures: introduction to engineering measurement, measurement theory, and software engineering measures. These materials include lecture notes suitable for class handouts and additional information for instructors—educational objectives, pedagogical considerations, suggestions for class projects, an annotated bibliography, and transparency masters for use in the delivery of the lectures.

Preface

Measurement is a fundamental skill for engineers, including software engineers. Computer science programs, however, frequently do not teach either engineering measurement in general or software engineering measurement in particular. This omission can be attributed, at least in part, to three problems: the absence of the material from most undergraduate computer science textbooks, the lack of familiarity with the material on the part of instructors, and the newness of much of the knowledge about software engineering measurement.

This package provides material for three 60-minute introductory lectures on aspects of engineering measurement. These lectures can be used together or separately, and they can be used at almost any level in a curriculum. They provide a foundation for subsequent, more detailed study of software engineering measurement.

The package has been designed to address the three problems identified in the first paragraph. To augment existing textbooks and to help instructors become familiar with software engineering measurement, the package includes three short expository documents, or "lecture notes":

- Introduction to Engineering Measurement
- Measurement Theory for Software Engineers
- Software Engineering Measures

The third of these documents, in particular, includes material that first appeared in the literature in 1992, and therefore has not yet been widely disseminated.

The package begins with information for instructors. It includes educational objectives for the lectures, recommendations for using the materials, pedagogical considerations,

suggestions for class exercises and projects, answers to selected discussion questions from the lecture notes, and an annotated reading list.

The second portion of the package contains the three lecture notes documents. Each is a stand-alone document intended to be photocopied and distributed to the students. Throughout the lecture notes are several discussion questions, research questions, and ideas for individual or class projects. We hope that these will help instructors engage the students in learning the material.

The next portion of the package contains masters for making overhead transparencies. These include many of the figures from the lecture notes, along with some of the discussion questions and other material we thought might be useful in delivering the lectures.

Finally, there are the detailed forms discussed in the software engineering measures lecture. Although these could be used as transparency masters, the very detailed nature of the forms suggests that they should be photocopied and given to the students. Some of the discussion questions and one of the suggested class projects require the students to use the forms.

Information for Instructors

1. Objectives

The overall objective of the materials in this package is to give students a basic level of knowledge and understanding of measurement and its application to software engineering.

The objectives of the lecture "Introduction to Engineering Measurement" are to enable students to:

- understand and use the vocabulary of measurement, including the terms *measure*, *measurement*, *accuracy*, and *precision*;
- recognize everyday examples of measurement in the physical world, and relate those measurements to engineering;
- explain in general terms what engineers measure, why they measure, and how they measure;
- explain the distinctions between product measures and process measures, between static measures and dynamic measures, and between direct measures and derived measures.

The objectives of the lecture "Measurement Theory for Software Engineers" are to enable students to:

- understand the difference between a *measure* and a *metric*, and to use both terms correctly;
- understand the measurement theory concepts of *relational system*, *scale*, *admissible transformation*, and *meaningful*;
- explain how measurement can be used to reason about objects and relationships in the physical world when direct reasoning fails;
- understand the *nominal*, *ordinal*, *interval*, *ratio*, and *absolute* classes of measurement scales, and explain the limitations imposed by each on the kinds of meaningful statements that can be made about measures in each class.

The objectives of the lecture "Software Engineering Measures" are to enable students to:

- understand the similarities and differences between software engineering measurement and measurement in the traditional engineering disciplines;
- explain what *can* be measured and what *should* be measured by software engineers, and why the two are not necessarily the same things;
- describe in general terms the measures of software *size*, *effort*, *schedule*, *quality*, *performance*, *reliability*, and *complexity*;

- describe important software attributes that we do not yet know how to measure;
- explain and use the SEI checklists for defining precise measurement of software size, effort, and defect counts;
- explain the importance of, and give examples of, quantitative measurable software requirements.

2. Where in the Curriculum to Use the Materials

These materials may be used in an undergraduate computer science curriculum at any level. The introduction to engineering measurement can be used in conjunction with any course that has a laboratory component, because a lab is an ideal environment in which to learn to perform measurement. See [Northrop93] for more on measurement in laboratories. The material on software engineering measures is appropriate in any course in which the students are doing large programming projects, especially team projects. The suggested class exercises fit well with such projects.

The lecture on measurement theory requires that the students be able to read mathematical notation, so it probably should be used after they have had a good calculus or discrete mathematics course.

Although the three lectures are closely related, they can be delivered individually. The instructor may need to provide some additional material or vocabulary from the other lectures, but there is not a strong dependency of any lecture on any other. However, if the introduction to engineering measurement is not followed relatively closely by the software engineering measures lecture, the instructor should develop some additional examples of measurement that are relevant to the current course.

3. Pedagogical Considerations

Engineering education should prepare students to be inquisitive and inventive—to be able to discover and construct new knowledge when it is needed. This requires the instructor to rely less on pure lecture and more on guided discussion and experiment.

The materials in this package include more than 20 discussion and research questions for the students. These will help the instructor engage the students in the learning process. We recommend that instructors use as many questions as possible, either in class, in labs, or as homework assignments. Ideally, the instructor can use them to help the students relate the measurement concepts to their everyday lives, and to see parallels between the engineering of software and the engineering of everyday products. Seeing these relationships helps the students remember and understand the concepts.

Suggested answers to most of the discussion questions are included in Section 6 of this document. However, for most questions, there is no one right answer. The instructor can use the suggested answers as a starting point, but should guide the students in exploring a range of answers.

The research questions are distinguished from the discussion questions in that they probably will require the students to go to the library to look up answers. These questions are usually tangential to the main ideas of the lecture, so they can be omitted. If an instructor uses them, it is appropriate to ask several students each to answer a part of the question (see, for example, research question 6 in "Introduction to Engineering Measurement"). Answers from each student can be distributed to all other students, either on paper or through electronic mail or a class electronic bulletin board, if available.

There is another aspect of engineering education that is sometimes overlooked: students of engineering should gain an understanding of the role of engineering in society. Unlike science, which can be done somewhat in isolation, engineering builds products for people. Students' understanding can be enhanced in many ways. One is to choose examples of engineering that are very familiar to the students as people and not just as engineers. A second is to reduce the compartmentalization of the subject matter of courses—engineering instructors should feel comfortable talking about the humanities, arts, or social sciences where appropriate in engineering courses; humanities instructors should feel comfortable talking about math or science where appropriate in their courses. Toward this end, these materials include some mention of history and etymology.

4. Suggestions for Class Exercises and Projects

The nature of engineering requires people to work in teams, so class exercises and projects are an important part of engineering education. The material in the lecture on software engineering measures fits well in a project-oriented course, and it also suggests some useful software projects.

Two class exercises are included in the lecture notes (and reproduced below). These are short exercises that require the whole class to participate and, thus, can be given as homework assignments.

The objective of the first exercise is to convince the students that counting lines of code is not as easy as it sounds. It is likely that the counts of physical lines of code will be more consistent than those of logical lines of code. The instructor can ask first for a count of the number of "lines of code" without specifying physical or logical, in order to increase the variance in student answers and thus increase the impact of the exercise.

This exercise can be conducted in class. The instructor may wish to bring blank transparencies and markers to class so that the histograms can be created immediately after the students give their counts.

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, page 3 of "Software Engineering Measures").

```

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;

```

The second exercise addresses the common concern that software size measured in logical lines of code is somehow better than physical lines of code, by showing that the two measures are related. Prior to discussing this exercise, instructors may want to read Section 3.2.1 in [Carleton92], which presents the rationale for the SEI recommendation to use physical rather than logical lines of code as a size measure.

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 checklists 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.

This exercise works only if all students are using the same size definition checklists. The instructor can develop an appropriate checklist in class, based on recommendations from the students. Then the students can apply the checklist to their programs as a homework assignment. Either the instructor or a designated student can collect the data from all students and look for the mathematical relationship.

Instructors should note that doing these exercises in class will take a significant amount of time, so it would be wise to allocate more than 60 minutes to covering the material on software engineering measures.

The following class project is included in the lecture notes. It might be described as a measurement-related "add-on" to a large programming project that is already part of the course. It is intended to give the students a taste of the professional software engineering environment.

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 development phases 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.

There are also programming projects that build software tools to support measurement. An obvious example is a tool that can measure lines of code. The items on the SEI definition checklist are parameters that can be varied. A design goal should be that the tool be easily modifiable to work on different programming languages; thus, language-specific code should be minimized and encapsulated in a module.

Another programming project is a database that holds size data in the categories on the SEI definition checklist. The program should be able to produce the kinds of reports defined by various data array specifications.

5. Suggested Answers to Discussion Questions

The discussion questions in the lecture notes are reproduced here for the benefit of instructors. We have included a suggested answer or partial answer for each. In general, there is no single, complete, correct answer. We hope the answers given will help instructors conduct a classroom discussion; this is an important and effective way of teaching much of the measurement material.

5.1. Questions from "Introduction to Engineering Measurement"

Discussion Question 1

Measurement of length almost certainly predates historical records. The earliest measures were probably in terms of the human body, and some of those measures survive to this day. The most obvious example is the *foot*. What are some other such measures? (This question may be easier if you have had occasion to measure horses or whiskey.) What is a *cubit*?

Answer

Horses are measured in *hands* and a glass of whiskey is sometimes measured in *fingers*. A *cubit* is the distance from the elbow to the end of the outstretched middle finger, typically about 18 inches.

Discussion Question 2

What are some common units of measure that use the prefixes in Figure 1 (page 3 of "Introduction to Engineering Measurement")? What is another term for one one-millionth of a meter, and why is a machinist likely to prefer it to *micrometer*? Why is the term *decibel*, a unit of loudness, much more common than the whole unit, the *bel*?

Answer

Some common units of measure are kilobyte, kilometer, and kilogram; megabyte; decibel; centimeter; millisecond, millimeter, and milligram; and microsecond. One one-millionth of a meter is commonly called a *micron*. A machinist uses a tool called a *micrometer caliper*, or more commonly, a micrometer. The term *decibel* may be more common because the loudness of sounds we hear in everyday life is in the range of about 50 to 100 decibels, and we may be more comfortable dealing with these whole numbers than with measures like 6.2 and 7.3 bels.

Research Question 3

What reasoning might have been used to choose the names of the prefixes in the metric system? Do the words mean anything? Hint: What are the Greek words for *ten*, *hundred*, and *thousand*? What are the Latin words? What are the Danish or Norwegian words for *fifteen* and *eighteen*? What is an Italian word for *small*? What are Greek words for *small*, *large*, *giant*, *dwarf*, and *monster*?

Answers

Latin: *decem* (ten), *centum* (hundred), *mille* (thousand). This also suggests the origin of the English word *mile*, which originally meant the length of 1000 double steps by a Roman soldier.

Greek: *deka* (ten), *hekatón* (hundred), *chilioi* (thousand), *mikros* (small), *megas* (large), *gigas* (giant), *nanos* (dwarf), *teras* (monster).

Danish and Norwegian: *femten* (fifteen), *atten* (eighteen).

Italian: *piccolo* (small).

Research Question 4

What do the terms *megaflops* and *gigalips* denote? Hint: These do not refer to Hollywood movies that lose millions of dollars or to a medical condition. Another hint: They do refer to computer performance.

Answers

The term *megaflops* means "million floating point operations per second" and is commonly used as a unit of measure for computers that perform scientific calculations. The term *gigalips* means "billion logical inferences per second" and is not so commonly used as a unit of measure for computers designed for artificial intelligence applications.

Discussion Question 5

What are some real-world entities that are measured in units using some of the more extreme prefixes? For example, is a typical human life span closer to a megasecond, gigasecond, or terasecond? How far does light travel in a microsecond, a nanosecond, or a picosecond? What two places are about a megameter apart? A terameter apart? Which is larger, a zettameter or the diameter of the Milky Way galaxy? Is the mass of an electron more or less than a yoctogram?

Answer

A human life span of 75 years is 236,675,520 seconds, or about one-quarter gigasecond. Light travels about 983 feet in a microsecond, 11.8 inches in a nanosecond, and about the thickness of three sheets of paper in a picosecond. The distance from New York to Charlotte, North Carolina, is about a megameter. The distance from the sun to Saturn is about 1.4 terameters. The diameter of the Milky Way galaxy is about one zettameter. The mass of an electron is about 0.001 yoctogram.

Research Question 6

The last four centuries have produced many scientists who made important contributions to our understanding of the physical world, and several of these scientists have been honored by having units of measure named for them. Identify the following scientists, the unit (or scale) of measure named for them, the kind of measure it is, and its definition in terms of the fundamental measures.

Answer

See the table on the next two pages.

Discussion Question 7

What measures of the current state of your world do you make periodically? What trends are you trying to identify?

Answer

You may want to suggest to the students such measures as weight (especially for dieters), bank balance, and grade point average. Athletes in training track their performance. We may notice an odometer reading periodically on a trip in order to determine average speed and predict our arrival time. We may notice our car's fuel gauge or a home heating oil measurement to predict when we will need to buy more fuel. We may watch the price of stocks to know when to buy or sell.

Discussion Question 8

How can we as software engineers rephrase these requirements in quantifiable—and therefore potentially measurable—terms?

Answer

Performance measurements vary widely with the application. We might say that a compiler must compile 2000 lines per minute, or a word processor must open a document in 2 seconds or scroll a whole page in one-half second. We might require that the object code for the controller in a VCR fit within 4 kilobytes of storage.

This question is discussed in more detail in the lecture notes document "Software Engineering Measures."

Discussion Question 9

Have you had to write a term paper or a major computer program and discovered the night before it was due that you still had 50% or 80% of the work ahead of you? Assuming that the problem was not just procrastination, how might you have been helped by a realistic schedule backed up by quantitative progress measurements?

Answer

There is no single answer to this question. Good estimates of the amount of work needed on a project may help you choose a project that can be completed in the allotted time. Early detection of slippages in schedules may permit adjustments in the schedule or different approaches to the work that will result in timely completion of the project.

Name	Identification	Unit or Scale	Definition
André-Marie Ampère	French physicist 1775-1836	<i>ampere</i> : electric current	one coulomb per second, or current produced by one volt across one ohm
Anders J. Ångström	Swedish physicist 1814-1874	<i>angstrom</i> : length	10^{-10} meter
Amedeo Avogadro	Italian chemist, physicist 1776-1856	<i>Avogadro's number</i> : number of atoms or molecules in a mole	6.023×10^{23}
Alexander Graham Bell	American inventor 1847-1922	<i>bel</i> : ratio of electric or acoustical signal power	$\log p_1/p_2$
Anders Celsius	Swedish astronomer 1701-1744	<i>Celsius</i> : temperature scale	
Charles A. de Coulomb	French physicist 1736-1806	<i>coulomb</i> : electric charge	quantity of charge transferred by one ampere in one second
Marie Curie and Pierre Curie	French chemists 1867-1934, 1859-1906	<i>curie</i> : radioactivity	3.7×10^{10} disintegrations per second
Gabriel D. Fahrenheit	German physicist 1686-1736	<i>Fahrenheit</i> : temperature scale	
Michael Faraday	English chemist and physicist 1791-1867	<i>faraday</i> : quantity of electricity	quantity transferred in electrolysis per equivalent weight of an element (approx. 96,500 coulombs)
		<i>farad</i> : capacitance	capacitance of a capacitor with one volt potential when charged by one coulomb
Enrico Fermi	Italian/American physicist 1901-1954	<i>fermi</i> : length	10^{-15} meter
Karl Friedrich Gauss	German mathematician, astronomer 1777-1855	<i>gauss</i> : magnetic flux density	10^{-4} tesla
Joseph Henry	American physicist 1797-1878	<i>henry</i> : inductance	inductance of a circuit in which the variation of one ampere per second results in an induced electromotive force of one volt
Heinrich R. Hertz	German physicist 1857-1894	<i>hertz</i> : frequency	one cycle per second; or second^{-1}
James P. Joule	English physicist 1818-1889	<i>joule</i> : work or energy	10^7 ergs
William Thomson, Lord Kelvin	English mathematician, physicist 1824-1907	<i>Kelvin</i> : temperature scale; <i>kelvin</i> : thermodynamic temperature	

Suggested Answers to Research Question 6

Name	Identification	Unit or Scale	Definition
James Clerk Maxwell	Scottish physicist 1831-1879	<i>maxwell</i> : magnetic flux	flux per square centimeter of normal cross section in a region where the magnetic induction is one gauss
Friedrich Mohs	German mineralogist ?-1839	<i>Mohs scale</i> : mineral hardness scale	
Isaac Newton	English mathematician, physicist 1642-1727	<i>newton</i> : force	1 kilogram per second per second
Georg Simon Ohm	German physicist 1787-1854	<i>ohm</i> : resistance	resistance of a circuit in which a potential difference of one volt produces a current of one ampere
		<i>mho</i> : conductivity	ohm^{-1}
Hans Christian Ørsted	Danish physicist, chemist 1777-1851	<i>oersted</i> : magnetic intensity	intensity of a magnetic field in a vacuum in which a unit magnetic pole experiences a mechanical force of one dyne in the direction of the field
Blaise Pascal	French mathematician, philosopher 1623-1662	<i>pascal</i> : pressure	1 newton per square meter
Charles R. Richter	American seismologist 1900-1985	<i>Richter scale</i> : earthquake intensity scale	
Wilhelm Röntgen	German physicist 1845-1923	<i>roentgen</i> : x-radiation or gamma radiation	amount of radiation that produces, in one cubic centimeter of dry air at 0°C and standard atmospheric pressure, ionization of either sign equal to one electrostatic unit of charge
Nikola Tesla	American physicist 1856-1943	<i>tesla</i> : magnetic flux density	1 weber per square meter
Allesandro Volta	Italian physicist 1745-1827	<i>volt</i> : electromotive force; electrical potential difference	potential across one ohm when one ampere of current is flowing
James Watt	Scottish inventor 1736-1819	<i>watt</i> : power	one joule per second; one volt times one ampere
Wilhelm E. Weber	German physicist 1804-1891	<i>weber</i> : magnetic flux	10^8 maxwells

Suggested Answers to Research Question 6 (continued)

Discussion Question 10

The classic tradeoff in programming is *time* vs. *space*. What does this mean? What are some examples? Can you describe a situation from your own experience in which you consciously made a time/space tradeoff?

Answer

Often there are several algorithms that will accomplish a particular task. Some of them may be faster but require more space. For example, some sorting algorithms may be fast but require an amount of temporary storage proportional to the size of the data being sorted; others are slower but need only a constant amount of temporary storage. Some algorithms require repeated calculation of particular intermediate values. If sufficient storage is available, we can compute the values once and save them; otherwise, we must recalculate them each time they are needed. When designing animation software, if sufficient memory is available, we may be able to create several different images beforehand, store them, and move them to the screen display memory rapidly when needed. Otherwise, the images may have to be recreated whenever needed, resulting in slower animation.

Discussion Question 11

What are some common instruments that you use to measure the following quantities? Estimate the accuracy and precision of the instruments. What kinds of errors are common in these measurements?

Answer

Your height: a tape measure or yardstick; accuracy and precision depend on the user, but are probably $\pm 1/8$ inch. Parallax errors are common.

Your weight: a bathroom scale; accuracy perhaps ± 5 pounds; precision perhaps ± 1 pound. Null-point errors are common; parallax and hysteresis errors may occur. You may want to ask the students to describe an experiment to look for hysteresis errors; one such experiment would be to compare readings from getting on the scale yourself and from getting on with another person, who then steps off.

The distance you drive your car on a trip: odometer; accuracy is probably $\pm 5\%$, precision may be $\pm 1\%$. Calibration errors are likely to be the most significant source of error.

The pressure in your car's tires: pressure gauge; accuracy is perhaps ± 3 psi, precision may be ± 1 psi. Random errors may be the most common because of the difficulty of using most pressure gauges in a consistent manner.

A spark plug gap: a feeler gauge; accuracy and precision are perhaps ± 0.002 inch. Random errors are common.

The time it takes an athlete to run 100 meters: stopwatch; if used consistently, accuracy and precision are probably within 0.2 second. Random errors are probably the most common because of the variability of the user's reaction time.

The temperature of a beef roast: a meat thermometer; accuracy and precision are maybe $\pm 10^\circ\text{F}$. Calibration errors are probably most common.

The frequency of the middle C note on a piano: a tuning fork; accuracy and precision are maybe ± 5 Hz. Calibration and random errors are common.

The thickness of a piece of paper: micrometer calipers; accuracy and precision are perhaps ± 0.001 inch. Calibration errors are possible. The thickness of a single sheet of paper may be near the limit of sensitivity of the instrument. The students might also suggest measuring a

known number of sheets of paper, such as a ream, with a ruler and then computing the thickness of a single sheet; this can be quite accurate also.

Discussion Question 12

What measurement instruments do you use that you consciously calibrate from time to time? Can you think of an everyday measurement where a null-point systematic error might be introduced purposely? Have you ever experienced a parallax error while you (or your passenger) were reading your car's speedometer or other instrument? Did the speedometer appear to read higher or lower to the passenger? How does this depend on whether the needle is in front of the numbered scale or behind it?

Answer

It is common to calibrate a clock or wristwatch from time to time. Some people like to set the null point on their bathroom scales to something other than zero. The passenger will normally see a higher than actual speed if the scale is in front of the needle; lower otherwise. Note that this is based on the assumption that the scale increases from left to right and the car is a left-hand drive model.

Research Question 13

What is a *vernier* and how does it work? What is its intended effect on the accuracy or precision of a measurement?

Answer

A vernier is a short scale that is used in conjunction with a longer scale and is designed so that its reading is tenths or hundredths of the smallest division of the longer scale. It can increase the accuracy and precision of measurements considerably.

Discussion Question 14

A little-known fact is that there are 51,500,000 hairs on the average horse. Suggest a sampling technique that might have been used to discover this fact.

Answer

First, make measurements of the horse so you can compute the surface area. Then count the hairs in several representative areas in patches of perhaps a square centimeter. Multiply the average number of hairs per square centimeter by the area of the horse. All this may be facilitated by choosing a friendly and patient horse.

Discussion Question 15

Suppose you are the manager of an engineering project with 200 staff members. You want to measure how much staff time will be spent on meetings, administrative paperwork, library research, laboratory work, writing reports, and work at the computer over the next year. Suggest a sampling technique that might provide estimates of these numbers without waiting the whole year.

Answer

Choose a representative sample of the staff, meaning people at all levels and with all kinds of responsibilities. Measure how they spend their time one day a week for a few weeks. Then extrapolate to the whole staff for the whole year.

5.2. Questions from "Measurement Theory for Software Engineers"

Discussion Question 1

For each of the following sets of objects, suggest a measure and scale for those objects, and identify the class in which the scale belongs (nominal, ordinal, interval, ratio, absolute).

Answer

Mass of physical objects: grams (ratio).

Loudness of sounds: decibels (logarithmic scale; see comments below on earthquake intensity).

Brightness of lights: candela (ratio).

Human intelligence: IQ (ordinal).

Beauty of the paintings in a museum: perhaps with something like a scale from 1 to 10 (ordinal); many might argue that this is so subjective that a nominal scale might be the best we could do.

Kelvin scale of temperature: kelvins (ratio); the Kelvin scale is based on energy, so it is not just an interval scale like the Celsius and Fahrenheit temperature scales.

Size of a software system: physical lines of code (absolute).

Productivity of different assembly line workers: widgets produced per hour (ratio).

Productivity of different software engineers: lines of code produced and delivered per hour (ratio).

Cost of different models of automobiles: dollars (ratio).

Reliability of different models of automobiles: frequency of repair, measured in number of times in the shop per year (ordinal); some might argue that this is an interval or ratio scale, which is probably true in the strict numerical sense but not in the sense of the underlying concept of reliability.

Desirability of vacationing in each of the 50 states of the US: perhaps with something like a scale from 1 to 50 (ordinal); very subjective, as with beauty of paintings.

Earthquake intensity: Richter scale (ordinal scale if we just look at the numbers; however, this is actually a logarithmic ratio scale, so we have to take that into account in statements like "a level 8 earthquake is twice as strong as a level 4 earthquake" [not true]; "a level 8 earthquake is 10,000 times as strong as a level 4 earthquake" [true]).

Speed of different models of computer: MIPS, meaning "million instructions per second" (ratio).

User-friendliness of word-processing or spreadsheet software: a scale of 1 to 10 (ordinal); very subjective.

Discussion Question 2

The cost of objects is usually regarded as a measure that has a ratio scale; it is meaningful to talk about one automobile model being twice as expensive as another. On the other hand, attributes such as the quality of a car or the complexity of a software system may be measurable only with ordinal scales (or perhaps interval scales). An engineer is often called upon to make judgments in terms of *value*, which we might define as *quality per unit of cost*. For example, should you pay twice as much for twice the quality? Should you pay more or less for software that is more complex? What is "today's best value in a luxury automobile"? When you create a value measure by combining a cost measure on a ratio scale with a quality measure on an ordinal or interval scale, what kind of a scale do you get?

Answer

There is no simple answer to these questions, mostly because quality can be defined and measured in so many ways. You may want to ask students if the unit prices found in most supermarkets help customers measure value. A package twice as large at twice the cost may be the same value. Two packages of a product that are the same size but different brands may be priced differently. Is the cheaper one of higher value?

Usually the kind of scale created from quality and cost scales depends on the quality scale involved.

Research Question 3

How does the science of thermodynamics allow us to assert that the Kelvin scale of temperature is a ratio scale and not just an interval scale (like the Fahrenheit and Celsius scales)?

Answer

The Kelvin scale, which allows us to specify temperature in "kelvins," not "degrees Kelvin," is based on the amount of energy present in the substance being measured. Thus the numbers on the scale can start at 0 kelvins and are not arbitrarily related to the freezing or boiling of water.

5.3. Questions from "Software Engineering Measures"**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 of "Software Engineering Measures") as an example.

Answer

The relationship between the number of semicolons and the number of carriage returns varies according to programming style and somewhat among the three languages. In almost all cases, however, there is likely to be a linear relationship between the two measures. The biggest unknown factor is how the programmer writes comments.

Discussion Question 2

Look carefully at the SEI effort reporting checklist. 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?

Answer

On small projects, such as typical programming assignments in undergraduate classes, it may be quite difficult to distinguish design, coding, and testing because students typically switch from one activity to another several times per hour. On larger projects, including those most often undertaken by software engineers, the measurement can be easier. The different phases may take weeks, so it is usually easy to know which one you worked on today. Keeping track of who attended which meetings and how long the meetings lasted may be the responsibility of a support staff person. Some programming support tools can automatically keep track of time spent editing documents, editing code, compiling, or testing.

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!"

Answer

When using a software package, we often believe we hit a particular key or issued a particular command when in fact we did something else. In many such cases, there is no undeniable record of our action, so we cannot prove that a user mistake did or did not occur.

When the behavior of a software package is not what the user manual says, either the manual or the software can be wrong. Without seeing the software specification, there is no way to tell which one is in error. To the user, it will usually seem to be a software error because the user's expectation was based on what the manual said. For the software engineer, who may have the software specification, it is easier to determine where the fault lies. If the specification does not cover the particular situation, it may be tempting to change the manual to match the software rather than vice versa. However, the cost of printing and distributing revised manuals will have to be weighed against the cost of creating and distributing revised software.

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?

Answer

The designers and users of a telephone system may want to measure performance in terms of the time it takes for a call to go through (a response time measure) or the number of calls that can be completed per minute (a throughput measure). Airplane performance measures include cruising speed and rate of climb. Banking machine performance might be measured in response time to verify the user's identification number and the time to complete a transaction. A washing machine might be measured in minutes per load or loads per day. A water heater's performance is often measured in gallons per hour or in recovery time (time to reheat after all the hot water is replaced by cold water). Fast-food preparation machines might be measured in start-up time or units of food prepared per hour.

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?

Answer

The developers of a word processor could instrument the code to record the time before and after each use of the spell checker and the number of words checked. The speed could be determined from these values. A user of the system might use a wristwatch or stopwatch to measure the

apparent time used in spell-checking a document. Knowing this time and the number of words in the document would allow a less accurate and less precise measure of the speed.

Some other common response time measures are the time to perform a particular formatting operation, the time to scroll up or down a page, and the time to open or close a document.

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?

Answer

The two most obvious performance measures are the response time for a particular authorization request from one terminal and the number of authorizations per minute for the overall system. A response time of 10 to 15 seconds might be acceptable to users of the system. A large system might have a requirement to be able to process several hundred requests per minute.

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 (mean time between failures)? High MTBF? Of these, which have high and low MTTR (mean time to repair)? 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?

Answer

If we define failure as degradation of performance below a desirable level, then we might expect a low MTBF for the oil, air filter, oil filter, spark plugs, and PCV valve. These components are typically replaced during preventive maintenance, and they have low MTTR. Other components with MTBF near the low end of the scale might include fuses, coolant, radiator hoses, and tires. These are also relatively easy to replace. High MTBF items might include the frame and engine block.

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?

Answer

There is no easy answer to this question. Lehman discusses some of these issues [Lehman80].

As suggested, quality may be defined as freedom from defects and suitability for use. If the user never sees a defect, then, in one sense, there is no defect. (How is this different from the old question, "If a tree falls in a forest and no one is there to hear it, is there a sound?" What is the definition of *sound*?) How might you define *defect* to make this argument valid? What about terms like *latent defect* or *potential defect*?

When software is intended for direct use by a person, such as a word processor or spreadsheet, it is impossible to have verifiably correct requirements. Issues of quality must depend on the user's perception of freedom from defects and suitability for use, rather than a proof of those attributes.

Software that is embedded in a machine or system may be a different case. Consider, for example, the software that controls the operation of a VCR. There is only a small number of kinds and sequences of inputs, and the response to each input can be rigorously specified. Under those conditions, we might well expect to be able to prove the software correct.

Computer science sometimes tends toward the abstract, absolute end of the philosophical spectrum on issues like these, while software engineering tends toward the real-world, actual-use end of the spectrum.

Discussion Question 9

Although we cannot measure most of 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 that module only. 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*.

Answer

There are no specific answers to this question. However, students are likely to suggest ideas related to modularity, information hiding, and parameterization. This provides a basis for an argument that such programming practices can contribute to overall software quality, even though we can't measure a direct relationship.

6. Further Reading

Below is a short annotated bibliography of sources from which the three sets of lecture notes were derived. Instructors teaching this material for the first time may want to spend some time reviewing these references. Nearly all of them are readily available and quite readable.

Because the materials can be used in such a wide range of situations, we chose not to include bibliographies or suggested further readings in the lecture notes documents for

the students. Instead we have annotated this bibliography to give some guidance to the instructor with respect to which items might be appropriate for students at various levels. We recommend that instructors identify one or two items for each lecture, especially for the benefit of the better students.

- Carleton92 Carleton, A. D.; Park, R. E.; Goethert, W. B.; Florac, W. A.; Bailey, E. K.; & Pfleeger, S. L. *Software Measurement for DoD Systems: Recommendations for Initial Core Measures* (Tech. Rep. CMU/SEI-92-TR-19, ADA 258305). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.

Abstract: *This report presents our recommendations for a basic set of software measures that Department of Defense (DoD) organizations can use to help plan and manage the acquisition, development, and support of software systems. These recommendations are based on work that was initiated by the Software Metrics Definition Working Group and subsequently extended by the SEI to support the DoD Software Action Plan. The central theme is: the use of checklists to create and record structured measurement descriptions and reporting specifications. These checklists provide a mechanism for obtaining consistent measures from project to project and for communicating unambiguous measurement results.*

This report presents a summary of the recommended initial core measures that are detailed in three other SEI technical reports [Park92, Goethert92, Florac92]. It describes in general terms much of the motivation and justification for the recommended measures. It is good background for instructors, but much of it will be lost on students who have never experienced the industrial software environment.

- Dunham83 Dunham, J. R.; & Kruesi, E. "The Measurement Task Area." *Computer* 16, 11 (Nov. 1983): 47-54.

This paper provided some of the ideas on why engineers measure for the lecture on engineering measurement. It is good background reading for instructors and is probably readable by students at the junior or senior level.

- Florac92 Florac, W. A., et al. *Software Quality Measurement: A Framework for Counting Problems and Defects* (Tech. Rep. CMU/SEI-92-TR-22, ADA 258556). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.

Abstract: *This report presents mechanisms for describing and specifying two software measures—software problems and defects—used to understand and predict software product quality and software process efficacy. We propose a framework that integrates and gives structure to the discovery, reporting, and measurement of software problems and defects found by the primary problem and defect finding activities. Based on the framework, we identify and organize measurable attributes common to these activities. We show how to use the attributes with checklists and supporting forms to communicate the definitions and specifications for problem and defect measurements. We illustrate how the checklist and supporting forms can be used to reduce the misunderstanding of measurement results and can be applied to address the information needs of different users.*

This report presents in detail the ideas on software quality measurement introduced in [Carleton92]. It discusses why it is important to be able to measure software problems and defects in terms of quality, cost, and schedule. The report will be most useful to instructors, but is also appropriate for students in courses or course segments that address software project management or quality assurance.

- Goethert92 Goethert, W. B., et al. *Software Effort Measurement: A Framework for Counting Staff-Hours* (Tech. Rep. CMU/SEI-92-TR-21, ADA 258279). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.

Abstract: *This report contains guidelines for defining, recording, and reporting staff-hours. In it we develop a framework for describing staff-hour definitions, and use that framework to construct operational methods for reducing misunderstandings in measurement results. We show how to employ the framework to resolve conflicting user needs, and we apply the methods to construct specifications for measuring staff-hours. We also address two different but related aspects of schedule measurement. One aspect concerns the dates of project milestones and deliverables, and the second concerns measures of progress. Examples of form for defining and reporting staff-hour and schedule measurements are illustrated.*

This report presents in detail the ideas on software quality measurement introduced in [Carleton92]. The report will be most useful to instructors, but is also appropriate for students in a course on software project management.

- Holman89 Holman, J. P. *Experimental Methods for Engineers, 5th Ed.* New York: McGraw-Hill, 1989.

This book presents many basic definitions related to engineering measurement, information on the design of experiments and the analysis of experimental data, and a very thorough discussion of instruments and techniques for measuring physical properties. It is probably most appropriate for students of mechanical engineering, but students in other engineering disciplines can benefit from it as well.

- IEEE83 *IEEE Standard Glossary of Software Engineering Terminology* (ANSI/IEEE Std 729-1983). New York: IEEE, 1983.

The definitions of the *ilities* in the lecture on software engineering measures takes its definitions from this document. It is a useful reference for both instructors and students of software engineering.

- Lehman80 Lehman, M. M. "Programs, Life Cycles, and Laws of Software Evolution." *Proceedings of the IEEE* 68, 9 (Sept. 1980): 1060-1076.

Abstract: *By classifying programs according to their relationship to the environment in which they are executed, the paper identifies the sources of evolutionary pressure on computer applications and programs and shows why this results in a process of never ending maintenance activity. The resultant life cycle processes are then briefly discussed. The paper then introduces laws of Program Evolution that have been formulated following*

quantitative studies of the evolution of a number of different systems. Finally an example is provided of the application of Evolution Dynamics models to program release planning.

This paper provides the motivation for discussion question 8 in the lecture on software engineering measures. It also provides some motivation for measurement and its role in software maintenance.

- Lehman91 Lehman, M. M. "Software Engineering, the Software Process and Their Support." *Software Engineering Journal* 6 (Sept. 1991): 243-258.

Abstract: *Computers are being applied more and more widely, penetrating ever deeper into the very fabric of society. Mankind is becoming increasingly dependent on the availability of software and its continuing validity. To achieve this consistently and reliably, in an operational domain that is forever changing, requires disciplined execution of the software development and evolution process and its effective management. That is the goal of advanced software engineering. This paper summarises basic concepts of software engineering and of the software development process. This leads to a principle of uncertainty, analysis of its implications for the software development process, an overview of computer-assisted software engineering (CASE) and brief comments on the societal relevance of these topics. For researchers in the field and practitioners familiar with individual concepts, issues and specific solutions, the paper provides a unifying framework, a basis for conceptual advance. Those without a significant practical software engineering background and experienced graduate students will extend general familiarity with fresh insights, new concepts and additional detail. Undergraduate and graduate students without significant experience may treat the paper as an introductory text.*

This paper and [Lehman80] both provide a wealth of ideas about what software engineering is and how measurement can play an important role. The author makes the point that the major success of measurement is not in measuring products after they have been built, but in providing models and mechanisms for analysis and forecasting. Both papers can be read by advanced undergraduate students; both should be read by instructors.

- Musa87 Musa, J. D.; Iannino, A.; & Okumoto, K. *Software Reliability: Measurement, Prediction, Application*. New York: McGraw-Hill, 1987.

A graduate course in software reliability is the best place to use this book, but an instructor of undergraduates might be able to use it for background as well.

- Musa93 Musa, J. D. "Operational Profiles in Software-Reliability Engineering." *IEEE Software* 10, 2 (Mar. 1993): 14-32.

This paper clearly discusses the role of operational profiles in the determination of software reliability. It is useful background for instructors and it can be read by advanced undergraduate students.

- Northrop93 Northrop, L. M. *Experimental Methods for Software Engineers* (Educational Materials CMU/SEI-93-EM-10). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1993 (forthcoming).
- The materials in this package can assist instructors in the development of laboratories for undergraduate courses in both computer science and software engineering. Northrop suggests ideas for measurement laboratories and for teaching the role of measurement in experimentation.
- Park92 Park, R. E., et al. *Software Size Measurement: A Framework for Counting Source Statements* (Tech. Rep. CMU/SEI-92-TR-20, ADA 258304). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.
- Abstract:** *This report presents guidelines for defining, recording, and reporting two frequently used measures of software size—physical source lines and logical source statements. We propose a general framework for constructing size definitions and use it to derive operational methods for reducing misunderstandings in measurement results. We show how the methods can be applied to address the information needs of different users while maintaining a common definition of software size.*
- This report presents in detail the ideas on software size measurement introduced in [Carleton92].
- Parnas90 Parnas, D. L.; van Schouwen, A. J.; & Kwan, S. P. "Evaluation of Safety-Critical Software." *Communications of the ACM* 33, 6 (June 1990): 636-648.
- Instructors and students with a knowledge of basic probability and statistics should find this paper readable and useful. It contains a good introductory discussion of software reliability and reliability measurement. It distinguishes reliability, availability, and trustworthiness of software systems.
- Smith90 Smith, C. U. *Performance Engineering of Software Systems*. Reading, Mass.: Addison-Wesley, 1990.
- Although this book contains advanced material suitable for practitioners and graduate students, it can be useful to instructors who are preparing lectures for undergraduate courses. Chapter 7 discusses performance measurement, including many basic concepts.
- Zuse91 Zuse, H. *Software Complexity: Measures and Methods*. Berlin: Walter de Gruyter, 1991.
- This book probably has the most comprehensive presentation of software complexity measures currently available. It defines, categorizes, and discusses nearly 100 different measures. It also presents fundamentals of measurement theory. It can be useful to instructors, but it is too detailed for undergraduate students.

Lecture Notes

Introduction to Engineering Measurement

Measurement Theory for Software Engineers

Software Engineering Measures

Introduction to Engineering Measurement

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

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. At the heart of these questions is one of the most important concepts in engineering: measurement. An engineer needs to know why to make measurements, what can be measured, what should be measured, how to measure, and what to do with the results. Let's explore these issues in the general context of science and engineering.

1. Definitions

The first question might be, "What is measurement?" In its simplest form, we can think of measurement as associating a numeric value with an object or action. We interpret that value as the amount of some quality or attribute possessed by that object or action.

If we look in dictionaries and technical glossaries, we find definitions like these:

measure	(verb) To ascertain the quantity, mass, extent, or degree of something in terms of a standard unit or fixed amount, usually by means of an instrument or process; to compute the size of something from dimensional measurements; to estimate the extent, strength, worth, or character of something; to take measurements.
---------	---

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.

(noun) A standard or unit of measurement; the extent, dimensions, capacity, etc. of anything, especially as determined by a standard; an act or process of measuring; a result of measurement.

measurement The act or process of measuring something. Also a result, such as a figure expressing the extent or value that is obtained by measuring.

These definitions are very general and abstract. We can perhaps gain a better understanding of measurement by looking at examples from the everyday physical world.

2. Measures in the Physical World

The materials and forces of the physical world are the raw materials of the traditional engineering disciplines (such as civil, mechanical, electrical, and chemical engineering). Measuring in the physical world is thus a basic skill needed by engineers.

There are only a few fundamental physical measures. The most common of these are length, mass, and time. Other measures can be expressed in terms of these; for example, we express the speed of our cars in miles per hour (length divided by time), or our weight (which is really a measure of the force of gravity on our bodies at the earth's surface) in pounds (mass times length divided by time squared).

There can be many different units of measurement for a given physical measure. For example, we can measure length in meters, kilometers, inches, miles, and light-years, and we can measure time in seconds, minutes, hours, and years.

Discussion Question 1

Measurement of length almost certainly predates historical records. The earliest measures were probably in terms of the human body, and some of those measures survive to this day. The most obvious example is the *foot*. What are some other such measures? (This question may be easier if you have had occasion to measure horses or whiskey.) What is a *cubit*?

The most common set of measures is the *metric system*, used throughout the world, except in the United States, where the *English system* is more common. (Note that the English system of measures is no longer the official system in England!) The seven base units of measurement in the metric system are:

Unit	Entity Measured
meter	length
kilogram	mass
second	time
ampere	electric current
kelvin	thermodynamic temperature
mole	number of particles
candela	light intensity

A standard vocabulary of prefixes has been defined in order to measure very large and very small quantities. These are shown in Figure 1.

Prefix	Symbol	$\times 10^n$	Prefix	Symbol	$\times 10^n$
deka-	da	1	deci-	d	-1
hecto-	h	2	centi-	c	-2
kilo-	k	3	milli-	m	-3
mega-	M	6	micro-	μ	-6
giga-	G	9	nano-	n	-9
tera-	T	12	pico-	p	-12
peta-	P	15	femto-	f	-15
exa-	E	18	atto-	a	-18
zetta-	Z	21	zepto-	z	-21
yotta-	Y	24	yocto-	y	-24

Figure 1. Metric system measurement prefixes

Discussion Question 2

What are some common units of measure that use the prefixes in Figure 1? What is another term for one one-millionth of a meter, and why is a machinist likely to prefer it to *micrometer*? Why is the term *decibel*, a unit of loudness, much more common than the whole unit, the *bel*?

Research Question 3

What reasoning might have been used to choose the names of the prefixes in the metric system? Do the words mean anything? Hint: What are the Greek words for *ten*, *hundred*, and *thousand*? What are the Latin words? What are the Danish or Norwegian words for *fifteen* and *eighteen*? What is an Italian word for *small*? What are Greek words for *small*, *large*, *giant*, *dwarf*, and *monster*?

Research Question 4

What do the terms *megaflops* and *gigalips* denote? Hint: These do not refer to Hollywood movies that lose millions of dollars or to a medical condition. Another hint: They do refer to computer performance.

Discussion Question 5

What are some real-world entities that are measured in units using some of the more extreme prefixes? For example, is a typical human life span closer to a megasecond, gigasecond, or terasecond? How far does light travel in a microsecond, a nanosecond, or a picosecond? What two places are about a megameter apart? A terameter apart? Which is larger, a zettameter or the diameter of the Milky Way galaxy? Is the mass of an electron more or less than a yoctogram?

Digression. Helen of Troy is said to have been the most beautiful woman of the ancient world, and her abduction to Troy was the major cause of the Trojan War. The Greeks needed hundreds of ships to transport their soldiers to Troy, so it is sometimes said that Helen had the face that launched a thousand ships. Beauty is a difficult thing to measure, but the legend of Helen of Troy provides one possible solution. We can choose as our unit of measure the *millihelen*, which is defined as the precise amount of beauty necessary to launch exactly one ship.

In science and engineering, we usually choose to use the *mks* system (meter-kilogram-second), the *cgs* system (centimeter-gram-second), or the *fps* system (foot-pound-second). Each of these systems of measures includes a variety of other measures that can be expressed in terms of the basic measures.

Research Question 6

The last four centuries have produced many scientists who made important contributions to our understanding of the physical world, and several of these scientists have been honored by having units of measure named for them. Identify the following scientists, the unit (or scale) of measure named for them, the kind of measure it is, and its definition in terms of the fundamental measures.

André-Marie Ampère
Anders J. Ångström
Amedeo Avogadro
Alexander Graham Bell
Anders Celsius (scale)
Charles A. de Coulomb
Marie Curie and Pierre Curie
Gabriel D. Fahrenheit (scale)
Michael Faraday (2 answers)
Enrico Fermi
Karl Friedrich Gauss
Joseph Henry
Heinrich R. Hertz
James P. Joule

William Thomson, Lord Kelvin
James Clerk Maxwell
Friedrich Mohs (scale)
Isaac Newton
Hans Christian Ørsted
Georg Simon Ohm (2 answers)
Blaise Pascal
Charles R. Richter (scale)
Wilhelm Röntgen
Nikola Tesla
Allesandro Volta
James Watt
Wilhelm E. Weber

3. What Engineers Measure

Engineering is often described as being a process that results in useful products. It follows that we can describe what engineers measure in two broad categories: *product* measures and *process* measures.

We can further categorize the product measures as *static* and *dynamic*. Many of the physical measures of objects—such as size, length, height, width, weight, capacity, and volume—are static, meaning that they can usually be measured while the object is not in use. The dynamic measures describe the behavior of the object while it *is* in use; these include such attributes as velocity, fuel or power consumption, heat dissipation, vibration, and noise level. Engineers in the various disciplines (civil, mechanical, electrical, chemical, etc.) typically need to know dozens of specialized static and dynamic measures for the kinds of products they build.

The process measures are used to quantify the human activity of engineering, and they are much more alike across the various engineering disciplines than are the product measures. They typically include staff size, effort, calendar time, costs, and productivity. The importance of these measures is best understood by remembering that engineering activity in our society is subject to economic constraints. Whether the engineering is done by a private, profit-oriented company or by public, tax-supported engineers, the success of the project almost always depends on achieving the desired results on time and within budget.

We also characterize some measures as being *basic* or *directly measurable* quantities, and others as *composite* or *derived* quantities. Quantities like length, time, and weight are usually measured directly, while measures of productivity and velocity are often derived from direct measures by a mathematical operation (productivity can be computed by dividing the number of items produced by the time it took to produce them; velocity can be computed by dividing the distance traveled by the time it took).

4. Why Engineers Measure

There are several reasons why engineers measure. Let's look at the most important ones.

1. To describe the current state of the world

In one sense, every measurement describes an aspect of the *current* state of the world—a measurement made today describes something today, not how it was yesterday or how it will be tomorrow. But we know that things change over time, both in the physical world and in the software engineering world. If we can measure the current state of the world from time to time, it is often possible to discover *patterns* and *trends*.

The discovery of patterns in nature has always been one of the fundamental goals of science. Scientific explanations of the behavior of the physical world, what we often call *scientific theories* or *laws*, are almost always suggested by the results of measurements.

Once a theory has been formed, additional measurements can be made that support or refute the theory, thereby leading to a modified or improved theory.

A famous example of the relationships between theory and empirical measurement comes from the late 16th century. Johannes Kepler was attempting to describe the apparent motions of the planets with simple mathematical formulas, but without success. Only after he gained access to the Danish astronomer Tycho Brahe's substantially more accurate measurements of the positions of the planets did the ellipse suggest itself as the shape of the orbits. Kepler's theory provided a way of predicting the position of a planet at a specific time in the future, and other astronomers were able to make measurements that further supported the theory. Kepler's theory, when written as mathematical formulas, has come to be known as Kepler's laws.

Engineers are less often concerned with discovering the fundamental laws of nature than they are with discovering the behavior of the systems they design and build. They also may be concerned with trends in the engineering process itself. Measurements taken over time can help in both areas.

Discussion Question 7

What measures of the current state of your world do you make periodically? What trends are you trying to identify?

2. To state requirements quantitatively and demonstrate compliance

It is almost impossible to imagine an engineering project without quantitative requirements. A civil engineer designing a highway bridge over a river is concerned with the length of the bridge, the maximum traffic load, the height and flow of the river at flood stage, the maximum wind load the bridge must withstand, etc. An engineer designing appliances and small consumer products may be concerned with size, weight, cost, and power dissipation. An automotive engineer will have requirements of size, weight, power, passenger space, luggage space, emissions, and crash resistance. All of these requirements are likely to be expressed quantitatively. When the engineer wants to demonstrate that a product or system satisfies quantitative requirements, measurement is necessary.

Software engineers also work with quantitative requirements. Usually these describe the required performance of the system, although it is not uncommon to have requirements for the size of the object code (which may determine the number of integrated circuit chips required in an embedded system application) or the capacity of the system (such as in large information systems).

Computer programmers who are not software engineers often try to express requirements (either explicit or implicit) with phrases like "the program must be efficient," or "the program should be as fast as possible," or "the program must be memory-efficient." Because these requirements are not quantified, and probably not quantifiable, they are neither meaningful nor acceptable to a software engineer. The ability to measure offers

the engineer a way to state requirements quantitatively, and then to demonstrate compliance with those requirements.

Discussion Question 8

How can we as software engineers rephrase these requirements in quantifiable—and therefore potentially measurable—terms?

3. To track progress and predict results of an engineering project

In a large engineering project, periodic measurement of what has been accomplished or completed allows the project manager to track progress quantitatively. These measures can be especially useful in the identification of unusual trends, so the manager can foresee problems and try to solve them before they get out of hand. This can include not only technical problems, but also schedule or cost overruns.

Discussion Question 9

Have you had to write a term paper or a major computer program and discovered the night before it was due that you still had 50% or 80% of the work ahead of you? Assuming that the problem was not just procrastination, how might you have been helped by a realistic schedule backed up by quantitative progress measurements?

In many kinds of engineering projects, including software engineering, products undergo a period of testing and tuning before delivery. Measurements of product defects, breakdowns, or successful performance can be made over time, and then trends can be analyzed. Software engineers, for example, use defect counts during testing to calibrate reliability models, which in turn can predict when system testing will be complete and the desired level of system reliability achieved.

4. To analyze costs and benefits

Here we are getting at the heart of engineering: making tradeoffs. There are almost always many ways to design engineered products and many ways to design the components and subcomponents of those products. Each design offers advantages and disadvantages, and the engineer must trade one quality against another. Sometimes we are willing to accept more of a negative quality in order to get more of another, positive quality; sometimes we accept less of a desirable attribute in order to get more of another desirable attribute.

If we are automotive engineers, for example, we must make tradeoffs among weight, fuel economy, passenger room, ride comfort, and price. Will we accept more weight, and therefore decreased fuel economy, to gain more room or more ride comfort? Will we accept a higher showroom price to gain improved ride comfort? Will we accept the cost in time and money of research to develop a more fuel-efficient engine or a computer-controlled dynamic suspension system if it means we can provide both more comfort and better fuel economy?

To answer questions like these, engineers must have quantitative data on the costs and benefits of each design. That data comes from measurement.

Discussion Question 10

The classic tradeoff in programming is *time* vs. *space*. What does this mean? What are some examples? Can you describe a situation from your own experience in which you consciously made a time/space tradeoff?

5. How Engineers Measure

Traditional engineering measurement is performed with *instruments*. Much of the progress in science and engineering over the past few centuries has been facilitated by the development of new and better measuring instruments. But what makes an instrument *good*? Two very important considerations are accuracy and precision.

The *accuracy* of an instrument is an indication of how much the instrument's reading differs from a known input. Accuracy is usually expressed either as a percentage of the maximum measurable value, or as a range of deviation from a correct value. For example, if a voltmeter that can measure up to 1000 volts is said to be accurate to 1%, we know that any reading is no more than 10 volts (1% of 1000) high or low. We could also say that the voltmeter is accurate ± 10 volts (the symbol $\pm$ is pronounced "plus or minus").

The *precision* of an instrument is an indication of the repeatability of a measurement within a given accuracy. Suppose you measured a 120-volt source each day for a week with two different voltmeters, and you recorded these readings:

Day	Meter A	Meter B
Monday	125	120
Tuesday	126	117
Wednesday	125	123
Thursday	124	121
Friday	125	119

Meter A gave readings of 125 ± 1 volts, while meter B gave readings of 120 ± 3 volts. Meter A has better precision: ± 1 volt is better than ± 3 volts. However, we would also conclude that meter B is more accurate; it was never more than 3 volts off the correct value, while meter A was always 4 to 6 volts off the correct value.

An electrical engineer would also recognize that meter A could be *calibrated* so that its readings would have been 120 ± 1 volts, making it both more accurate and more precise than meter 2. Calibrating a voltmeter may be as simple as adjusting the value of an electrical component (such as a variable resistor) in the instrument. In general, cali-

bration of an instrument can improve its accuracy, but only up to the precision of the instrument. Meter B is probably already calibrated as well as it can be.

A calibration error is an example of a *systematic* error in measurement. These errors are a result of the physical limitations of instruments, and they occur in many forms. *Null-point* errors are caused when the zero or null reading on an instrument is improperly set, causing a constant shift in all readings. *Hysteresis* errors occur when a reading is influenced by the previous reading of the instrument, such as a voltmeter that gives a low reading when the needle approaches the correct value from below and a high reading when it approaches from above. *Parallax* errors occur when the measurements differ depending on the angle from the instrument to your eye.

Engineers are also concerned with *random* errors in measurement, which occur in the act of measurement itself, regardless of the systematic errors inherent in the instrument. Meaningful measurements are usually expressed not as a single value, but as a range in which the actual measurement lies (as in the example above, where the voltage measurement was 125 ± 1). This range is the *absolute* error in the measurement. The *relative* error is the absolute error divided by the measured value; it is usually expressed as a percentage.

Reducing systematic errors in measurement may require better instruments or better calibration of instruments. Random errors can be reduced by making several measurements and computing an average. A more precise study of random errors and their reduction depends on a good knowledge of statistics, so we won't pursue that here.

Discussion Question 11

What are some common instruments that you use to measure the following quantities? Estimate the accuracy and precision of the instruments. What kinds of errors are common in these measurements?

- Your height
- Your weight
- Your car's speed
- The distance you drive your car on a trip
- The pressure in your car's tires
- A spark plug gap
- The time it takes an athlete to run 100 meters
- The temperature of a beef roast
- The frequency of the middle C note on a piano
- The thickness of a piece of paper

Discussion Question 12

What measurement instruments do you use that you consciously calibrate from time to time? Can you think of an everyday measurement where a null-point systematic error might be introduced purposely? Have you ever experienced a parallax error while you (or your passenger) were reading your car's speedometer or other instrument? Did the speedometer appear to read higher or lower to the passenger? How does this depend on whether the needle is in front of the numbered scale or behind it?

Research Question 13

What is a *vernier* and how does it work? What is its intended effect on the accuracy or precision of a measurement?

Engineers (and scientists) often use another measurement technique: *sampling*. This may be defined as selecting and measuring a representative part of a population for the purpose of deducing parameters or characteristics of the whole population. It is used in situations where it is impossible or impractical to measure the whole population.

Discussion Question 14

A little-known fact is that there are 51,500,000 hairs on the average horse. Suggest a sampling technique that might have been used to discover this fact.

Discussion Question 15

Suppose you are the manager of an engineering project with 200 staff members. You want to measure how much staff time will be spent on meetings, administrative paperwork, library research, laboratory work, writing reports, and work at the computer over the next year. Suggest a sampling technique that might provide estimates of these numbers without waiting the whole year.

A related kind of engineering sampling is seen in *quality control* in the manufacturing process. At the factory assembly line, we may take every 100th or 1,000th product for testing and measurement. Using a variety of statistical techniques, we draw conclusions about the quality of all the products from the measurements made on the sampled products. Then we adjust the manufacturing process accordingly.

As was the case with measurement error, a thorough discussion of sampling techniques depends on knowledge of statistics, so we will not pursue it here.

6. Concluding Comments

Measurement is an important tool for engineers. As a student of engineering, you should learn:

- what can be measured
- what should be measured
- what kinds of instruments are used in measurement
- why measurements are needed
- how measurements are used
- how to measure the important quantities in the branch of engineering you are studying

Learning to measure requires practice, so laboratories and other "hands-on" experiences are useful. But pay attention also to the measurements you make in everyday life—they can teach you a lot about measurement.

Large engineering projects require many kinds of measurements and generate an enormous amount of data. Professional engineers need a good background in statistics in order to use that measurement data effectively. You would be well served by including a statistics course in your studies.

Measurement Theory for Software Engineers

Although mathematics might be considered the ultimate *abstract* science, it has always been motivated by concerns in the real, physical world. Thus it is not surprising that mathematics includes a branch called *measurement theory*. Some basic definitions from measurement theory can provide a more precise vocabulary for the study of software engineering measurement.

1. Measures and Metrics

Informally, we think of a measure as a way of associating a number, representing some attribute, with a physical object. Such an association is usually called a *mapping* or a *function* in mathematics. More formally, we can give this definition:

Definition 1. Let A be a set of physical or empirical objects. Let B be a set of formal objects, such as numbers. A *measure* μ (the Greek letter "mu") is defined to be a one-to-one mapping $\mu: A \rightarrow B$.

The requirement that the measure be a one-to-one mapping guarantees that every object has a measure, and every object has only one measure. It does *not* require that every number (in set B) be the measure of some object (in set A).

Another term that we use informally in measurement, and one that seems to appear frequently in the literature on software measurement, is *metric*. Some, but not all, measures are metrics. A metric is a way of measuring the distance (itself a term with many interpretations) between two entities, and it has this precise mathematical definition:

Definition 2. Let A be a set of objects, let R be the set of real numbers, and let $m: A \rightarrow R$ be a measure. Then m is a *metric* if and only if it satisfies these three properties:

$$m(x, y) = 0 \text{ for } x = y$$

$$m(x, y) = m(y, x) \text{ for all } x, y$$

$$m(x, z) \leq m(x, y) + m(y, z) \text{ for all } x, y, z$$

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.

We could actually allow other sets of numbers in this definition, as long as the set includes zero and the addition and less-than-or-equal operations are defined on the set.

A common example of a metric is the Euclidean distance metric in the plane. Let $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ be two points in the plane. Define the distance metric d by:

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Another is the "Manhattan" metric, so named because it is closer to the distance we would travel between two points in (an idealized) New York if constrained to move only along the east-west and north-south streets. For the same two points as above, we define this metric m by:

$$m(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$$

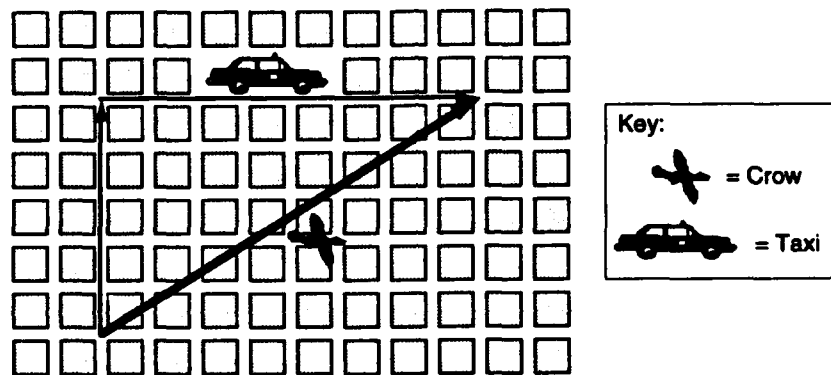


Figure 1. Euclidean distance and Manhattan metrics

Figure 1 illustrates these two metrics. To go five blocks north and eight blocks east "as the crow flies," the distance is approximately 9.43 blocks, but the distance "as the taxi drives" is 13 blocks.

From this definition we see that it is usually imprecise to speak of a "software metric." The preferred term is "software measure."

2. Measures and Relationships

The definition of measure is so general that some measures may not be particularly useful. For example, we can "measure" the members of a football team by the numbers on their jerseys. However, this measure is not very meaningful when used to answer a question like "Does player number 64 deserve twice the salary of player number 32?"

We are all familiar with measures of temperature. Suppose you looked at your Fahrenheit thermometer yesterday at noon and noted that the temperature was 80°, and today at noon it is only 40°. Is today half as warm as yesterday? If the temperature yesterday was only 2° and today it is 8°, is today four times as warm as yesterday?

Suppose your neighbor made the same observations but used a Celsius thermometer. Would the same questions make sense?

Some cities report an air quality index, a measure of the toxic gases and particulate matter in the air. Does an air quality index of 40 mean that the air is twice as unhealthy as an index of 20?

Questions like these assume that two objects have a relationship that can be understood by looking at a relationship between those object's measures. Using the right kind of measure may be critical to answering these questions meaningfully.

For some kinds of objects and measures, the relationships are clear. If one board is two feet long and another is three feet long, the second is longer; furthermore, the relationship between the "amount of board" is the same as the relationship between the measures (the numbers 2 and 3 in this case). Notice also that it is possible to select the longer board without measuring either, which indicates that we use an intuitive notion of "length of board" that is independent of any measure that might be applied.

Sometimes we can manipulate objects to get new objects, and we want to ask questions about the measure of the new object. If one pile contains 50 pounds of sand and another contains 100 pounds, when we combine the piles we expect to have 150 pounds of sand. We implicitly understand that the physical operation of combining piles has the corresponding operation of addition applied to the measures of the piles.

In other cases, the correspondence is anything but clear. Let us consider the *complexity* of a software system. Suppose we are building a software system and we have developed two possible designs. One breaks the system into 10 modules, each with complexity measure in the range of 20 to 30. The other design breaks the system into 20 modules, each with complexity measure in the range 10 to 30. Which design produces a better (less complex) overall system?

This last example illustrates an important aspect of all engineering problems: choosing among alternative solutions. Measurement can be very helpful in such situations, if we use appropriate measures. Most often there are several measures that can be made of the alternatives, and we will need to make *tradeoffs*; we may accept less of one desirable factor to get more of another desirable factor, or we may accept more of an undesirable factor to get more of a desirable one. But how do we use measures to know how much of one factor to trade off for how much of another factor? And how do we use measures to choose among alternatives?

3. Measures and Scales

The problems mentioned above—choosing the longer of two boards or recognizing that combining two piles of sand gives a larger pile of sand—are easy because we have intuitive meanings for the operations of "compare board length" and "combine sand piles." We do not have a corresponding intuitive meaning for the complexity of a software system based on the complexity of its components. We might describe this as an

“intelligence barrier” between questions about objects and the answers to those questions. This barrier is shown in the left side of Figure 2.

Measurement helps us answer these questions as shown in the right side of Figure 2. We first measure the objects in question, yielding (usually) numbers. Then we apply mathematical or statistical techniques to the numbers, yielding another number that somehow relates to the answer to the original question. The final step is *interpretation* of the result, yielding an answer in terms of the original object domain rather than in the domain of numbers.

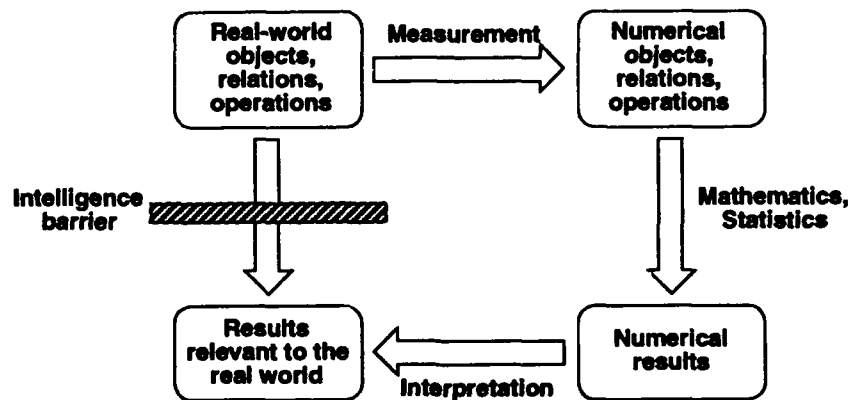


Figure 2. Measurement and the intelligence barrier

For example, suppose you were given a dozen boards of different lengths and asked to select the board of “average” length. It would most likely not be intuitively obvious which board to pick. However, if you measure the length of all the boards, you get a set of numbers. You can then compute the average (*arithmetic mean*) of those numbers. Finally, you interpret this number as the length of the board to be chosen, and pick the board closest to this length.

This technique works because there is an operation on boards that is appropriately modeled by the arithmetic mean operation on numbers. Measurement theory calls this relationship a *scale*.

Mathematically, we can give these definitions:

Definition 3. A *relational system* is defined as an ordered tuple $(S, rel_1, \dots, rel_n, op_1, \dots, op_m)$, where:

S is a nonempty set of objects;

$rel_1, \dots, rel_n$ are k_i -ary relations on objects in S (this means that the relation rel_i defines a relationship among k_i objects);

$op_1, \dots, op_m$ are binary operations on objects in S (this means that each operation operates on exactly two objects, producing a third object in S).

An example of a relational system of real-world objects is one with S being the set of all piles of sand, a binary relation "bigger than or same size as," and a binary operation "combine with." An example of a relational system of formal objects (in this case numbers) is one with S being the set of all nonnegative real numbers, the binary relation $\geq$, and the binary operation $+$.

In fact, these two examples can be shown to be, in some sense, the same if we apply the "weight in pounds" measure to piles of sand, yielding numbers, and we interpret those numbers as weights of piles of sand. Mathematically, we can make this definition:

Definition 4. Let $A = (S_A, relA_1, \dots, relA_n, opA_1, \dots, opA_m)$ be a relational system of physical or empirical objects, and let $B = (S_B, relB_1, \dots, relB_n, opB_1, \dots, opB_m)$ be a relational system of formal objects (such as numbers). Let $\mu: S_A \rightarrow S_B$ be a measure. Then the triple (A, B, μ) is a *scale* if and only if

$$relA_i(a_{i_1}, \dots, a_{i_k}) \Leftrightarrow relB_i(\mu(a_{i_1}), \dots, \mu(a_{i_k}))$$

and

$$\mu(a opA_j b) = \mu(a) opB_j \mu(b)$$

for all values of i and j , and for all $a, b, a_{i_1}, \dots, a_{i_k} \in S_A$.

More informally, this definition says two things. First, for every relation defined on the physical objects, there is a equivalent relation defined on the measures of those objects. By *equivalent*, we mean that if a statement about a relationship between or among objects is true, then the corresponding relationship between or among their measures is also true. Second, for every operation defined on the physical objects, there is a corresponding operation defined on the measures, such that the result of measuring the combined objects is the same as performing the corresponding operation on the measures of the individual objects.

A very mathematical note. The branch of mathematics known as *abstract algebra* deals with abstract entities consisting of sets of objects and associated operations. A mapping from one of these entities to another that preserves the operations in the way described in our definition of scale is called a *homomorphism*. This term comes from the Greek words for *same* (homos) and *form* (morphe). The measure μ in a scale is a homomorphism.

Mathematical measurement theory also helps us classify different kinds of scales and determine whether certain questions can meaningfully be asked and answered about objects measured with different kinds of scales.

To illustrate this, let's return to the discussion question posed earlier about the temperature yesterday and today. You noticed that the temperature yesterday was 80° and today it is 40°; you conclude that yesterday was warmer. Your neighbor, who has a Celsius scale thermometer, noticed it was about 27° yesterday and 4° today; she also concludes that yesterday was warmer.

This example suggests that it is meaningful to make statements such as "Yesterday was warmer than today" regardless of which temperature scale (Fahrenheit or Celsius) we are using. The intuitive concept of "warmer than" is preserved by measurement in the numeric concept "greater than" in both scales.

Now consider the more specific question posed earlier. Suppose you answered "yes" to the question, saying that today (40°) is half as warm as yesterday (80°). But your neighbor with the Celsius scale thermometer observed that today (4.4°) is only about one-sixth as warm as yesterday (26.7°). Who is correct?

This suggests that it is not meaningful to make statements such as "Yesterday was twice as warm as today" because the intuitive concept of "twice as warm" is not accurately reflected in the numeric concept "multiply by 2" applied to temperature measures. Different temperature scales give different results.

What is it about the two scales that make one kind of statement meaningful and another not? The Fahrenheit and Celsius temperature scales are closely related; there are simple algebraic expressions relating a temperature on one scale to a temperature on the other:

$$^{\circ}\text{F} = \frac{9}{5} ^{\circ}\text{C} + 32 \qquad ^{\circ}\text{C} = \frac{5(^{\circ}\text{F} - 32)}{9}$$

The relationship between the scales is *linear*, meaning it is of the form $f(x) = ax + b$. This is shown graphically in Figure 3. It is easy to see that what is "warmer" on one scale is also "warmer" on the other scale; this is because the coefficient a in $ax + b$ is positive. But it is also easy to see that "twice as warm" does not have the same meaning on both scales.

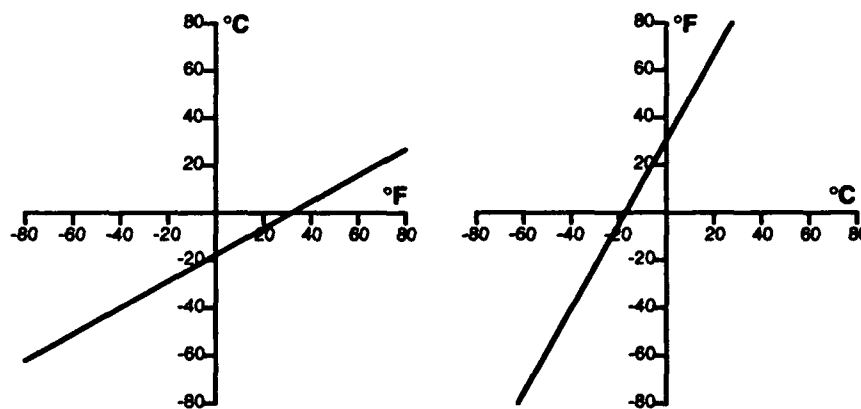


Figure 3. Linear relationships of Fahrenheit and Celsius scales

This example suggests that when two scales are related by a linear function (with a positive coefficient), then "greater than" and "less than" statements should give the same result (true or false) on both scales. A statement can be considered meaningful if it gives the same result on both scales.

Measurement theory allows us to formalize these ideas. First, we can formalize the relationship between two scales of measurement with this definition:

Definition 5. Let (A, B, μ) be a scale, where the set of objects in B is the set of real numbers. Let the notation $\mu(A)$ mean the set of all real numbers that are measures of some object in A . (In mathematics, we call this the *range of μ* .) Then a mapping $t: \mu(A) \rightarrow B$ is defined to be an *admissible transformation* if and only if the triple $(A, B, t \circ \mu)$ is also a scale.

We can interpret this definition as saying that if we have one scale of measure for a certain kind of object, we can invent other, equally good scales by applying admissible transformations to the original scale. Thus if we have the Fahrenheit scale for measuring temperature, we can invent the Celsius scale by applying the transformation $t(x) = 5/9x + 160/9$. If we have a scale of length in inches, we can invent a scale in centimeters by applying the transformation $t(x) = 2.54x$.

We now have a way of defining the meaningfulness of a statement made about the measures of objects:

Definition 6. Let (A, B, μ) be a scale, where the set of objects in B is the set of real numbers. A statement about the measures $\mu(a)$ of objects in A is *meaningful* if and only if the truth value (whether it is true or false) of that statement is unchanged after applying any admissible transformation to μ .

This definition requires, for example, that any meaningful statement made about the length of an object measured in inches should also be true if the object is measured in centimeters. If we have three boards as shown in Figure 4, then we can make statements such as "Board A is shorter than board B," or "Board B is twice as long as board C." These statements remain true if we measure the boards in centimeters instead of inches.

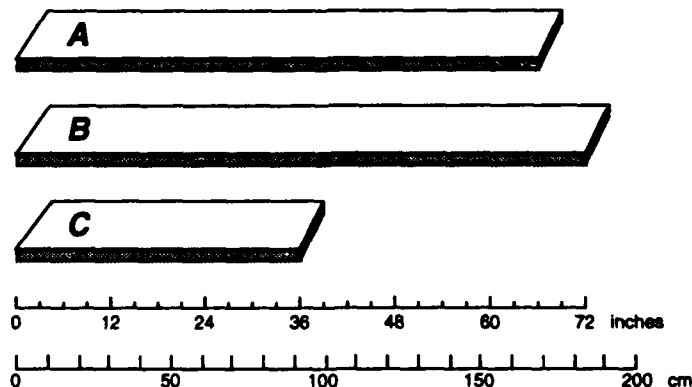


Figure 4. Boards measured in inches and centimeters

4. Classification of Scales

To finish our brief look at measurement theory, we want to consider the classification of scales and the kinds of admissible transformations that exist in each class. Throughout this discussion, we will assume that we are talking about a scale (A, B, μ) , where B is the set of real numbers, and transformations t .

Five kinds of scales can be described that are characterized by their admissible transformations:

Nominal scales simply give numeric "names" to objects. (The word *nominal* is derived from the Latin *nomin*, meaning *name*.) Any numbering is as good as any other, so any one-to-one function t is an admissible transformation. We have already mentioned one example of a nominal scale: the jersey numbers of football players. Any numbering of jerseys is as good as any other (except for other considerations, such as the convention that certain numbers represent certain positions on the team, or the fact that a 10-digit number would not fit on all but the very widest of players).

Ordinal scales assign numbers to objects in a particular order, but any numbers that maintain that order are equally good. Any strictly increasing function t is an admissible transformation. An example is the Mohs scale for the hardness of minerals. The original scale assigned, for example, 1 to talc, 7 to quartz, and 10 to diamond. Years later, a revised scale was created that assigned 1 to talc, 8 to quartz, and 15 to diamond. The numbers differed, but the order remained the same.

Interval scales assign numbers to objects in such a way that the interval between two measure values is meaningful throughout the range of values. Only positive linear functions $t(x) = ax + b$ are admissible transformations. We have already seen that the Fahrenheit and Celsius temperature scales are interval scales. A 10-degree difference between 20° and 30° and a 10-degree difference between 70° and 80° both mean the same thing with respect to how much heat is required to raise an object's temperature.

Ratio scales assign values in such a way that the ratio of two measures is meaningful. The only admissible transformations are positive linear functions of the form $t(x) = ax$. Length is a ratio scale, regardless of the unit of measurement, because ratio concepts like "twice as long" are meaningful.

Absolute scales have only one way of measuring objects, and so the only admissible transformation is the identity $t(x) = x$. Counting is the most common example of an absolute scale. Suppose we want to measure the staff size of a software project and make meaningful statements about the staff size. Counting the people is the obvious measure. We cannot imagine a transformation t other than the identity transformation that would make statements like "My project has 5 people on it" and "My project has $t(5)$ people on it" both true for all 5-person projects.

We should notice that this sequence of scales is increasingly restrictive. For example, every ordinal scale is also a nominal scale, but not vice versa. Every interval scale is

also an ordinal scale (and hence a nominal scale), but not vice versa. The relationships among the classes of scales is shown in the Venn diagram in Figure 5.

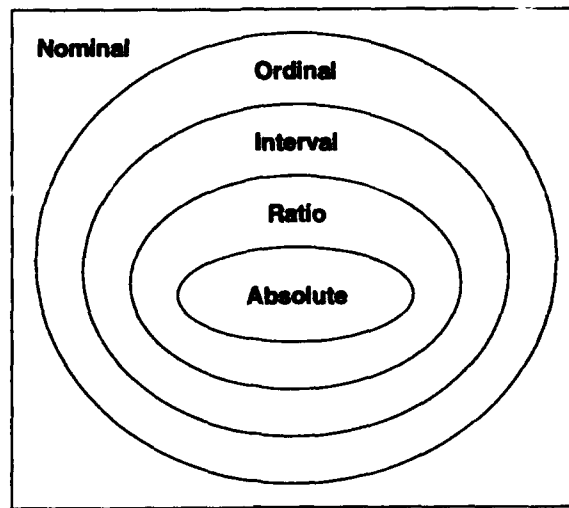


Figure 5. Relationships among classes of scales

5. Applying the Concepts of Measurement Theory

Our brief excursion into measurement theory teaches an important lesson for software engineering measurement: we should consider the kind of measurement scale we must have in order to make meaningful statements about our measurements. For example, if we want to say that one software system is twice as big or ten times as expensive as another, we need to be sure we have ratio scales for size and cost. If we want to talk about the average value of some measurements, we must have at least an interval scale. This lesson can be applied throughout our study of and practice of software engineering.

We can also use these ideas to conclude that it is not meaningful to make the statements "Today is half as warm as yesterday" and "Football player 64 is twice as good as player 32," because neither temperature nor jersey number is a ratio scale.

Discussion Question 1

For each of the following sets of objects, suggest a measure and scale for those objects, and identify the class in which the scale belongs (nominal, ordinal, interval, ratio, absolute).

- Mass of physical objects
- Loudness of sounds
- Brightness of lights
- Human intelligence
- Beauty of the paintings in a museum
- Kelvin scale of temperature
- Size of a software system

Discussion Question 1 (continued)

- Productivity of different assembly line workers
- Productivity of different software engineers
- Cost of different models of automobiles
- Reliability of different models of automobiles
- Desirability of vacationing in each of the 50 states of the US
- Earthquake intensity
- Speed of different models of computer
- User-friendliness of word-processing or spreadsheet software

Discussion Question 2

The cost of objects is usually regarded as a measure that has a ratio scale; it is meaningful to talk about one automobile model being twice as expensive as another. On the other hand, attributes such as the quality of a car or the complexity of a software system may be measurable only with ordinal scales (or perhaps interval scales). An engineer is often called upon to make judgments in terms of *value*, which we might define as *quality per unit of cost*. For example, should you pay twice as much for twice the quality? Should you pay more or less for software that is more complex? What is "today's best value in a luxury automobile"? When you create a value measure by combining a cost measure on a ratio scale with a quality measure on an ordinal or interval scale, what kind of a scale do you get?

Research Question 3

How does the science of thermodynamics allow us to assert that the Kelvin scale of temperature is a ratio scale and not just an interval scale (like the Fahrenheit and Celsius scales)?

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.

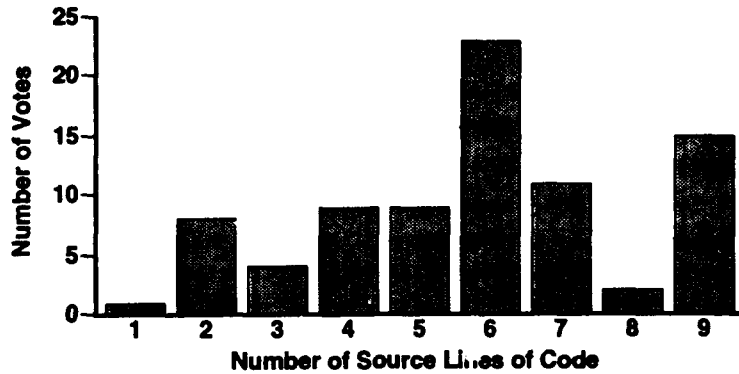


Figure 2. Results of the experiment

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 ☐	Includes	Excludes
<i>When a line or statement contains more than one type, classify it as the type with the highest precedence.</i>			
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: <u>Total System Staff-Hours</u> <u>for Development</u>	Date: <u>7/28/92</u> Originator: _____ Page: <u>1 of 3</u>		
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]	
	Include	Exclude	Value Count	Array Count
Problem Status				
Open	☒		☒	
Recognized				
Evaluated				☒
Resolved				☒
Closed	☒		☒	
Problem Type				
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				
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:

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

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*.

Classroom Materials

Transparency Masters

From "Introduction to Engineering Measurement"

Metric System (Figure 1)

Discussion Question (5)

Discussion Question (11)

From "Measurement Theory for Software Engineers"

Definitions (*measure* and *metric*)

Metrics (Figure 1)

Overcoming the Intelligence Barrier with Measurement (Figure 2)

Definition (*relational system*)

Definition (*scale*)

Definition (*admissible transformation*)

Definition (*meaningful*)

Meaningful Statements (Figure 4)

Discussion Question (1)

From "Software Engineering Measures"

Counting Lines of Code (Figures 1 and 2)

Class Exercise: How Many Lines of Code?

Definition Checklist for Source Statement Counts (Figure 3)

Staff-Hour Definition Checklist (Figure 6)

Problem Count Definition Checklist (Figure 8)

Software Measure Forms for Duplication

Definition Checklist for Source Statement Counts (4 pages)

Staff-Hour Definition Checklist (3 pages)

Problem Count Definition Checklist (2 pages)

Metric System

Prefix	Symbol	$\times 10^n$	Prefix	Symbol	$\times 10^n$
deka-	da	1	deci-	d	-1
hecto-	h	2	centi-	c	-2
kilo-	k	3	milli-	m	-3
mega-	M	6	micro-	μ	-6
giga-	G	9	nano-	n	-9
tera-	T	12	pico-	p	-12
peta-	P	15	femto-	f	-15
exa-	E	18	atto-	a	-18
zetta-	Z	21	zepto-	z	-21
yotta-	Y	24	yocto-	y	-24

Discussion Question

What are some real-world entities that are measured in units using some of the more extreme prefixes?

For example, is a typical human life span closer to a megasecond, gigasecond, or terasecond?

How far does light travel in a microsecond, a nanosecond, or a picosecond?

What two places are about a megameter apart? A terameter apart?

Which is larger, a zettameter or the diameter of the Milky Way galaxy?

Is the mass of an electron more or less than a yoctogram?

Discussion Question

What are some common instruments that you use to measure the following quantities? Estimate the accuracy and precision of the instruments. What kinds of errors are common in these measurements?

Your height

Your weight

Your car's speed

The distance you drive your car on a trip

The pressure in your car's tires

A spark plug gap

The time it takes an athlete to run 100 meters

The temperature of a beef roast

The frequency of the middle C note on a piano

The thickness of a piece of paper

Definitions

Let A be a set of physical or empirical objects. Let B be a set of formal objects, such as numbers. A *measure* μ is defined to be a one-to-one mapping $\mu: A \rightarrow B$.

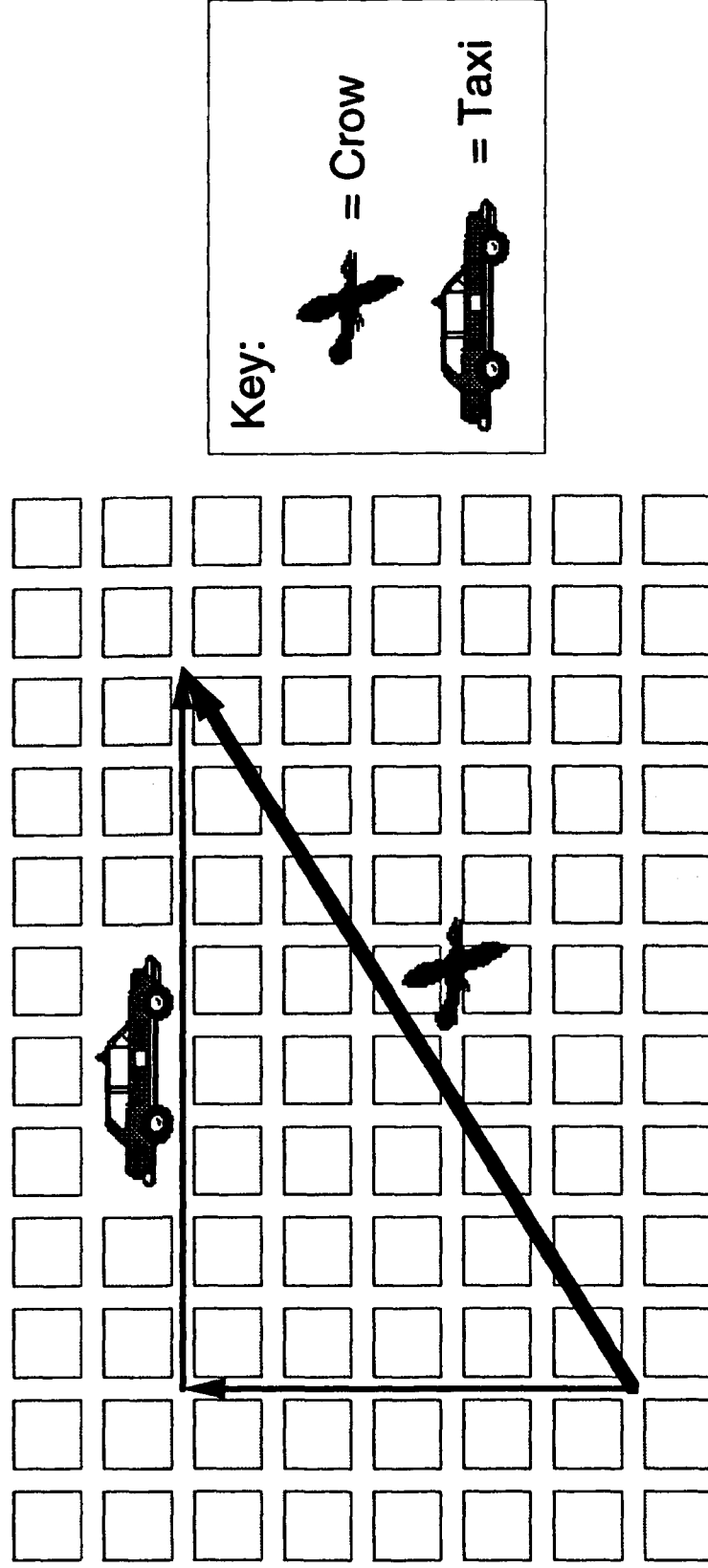
Let A be a set of objects, let R be the set of real numbers, and let $m: A \rightarrow R$ be a measure. Then m is a *metric* if and only if it satisfies these three properties:

$$\begin{aligned} m(x, y) &= 0 \text{ for } x = y \\ m(x, y) &= m(y, x) \text{ for all } x, y \\ m(x, z) &\leq m(x, y) + m(y, z) \text{ for all } x, y, z \end{aligned}$$

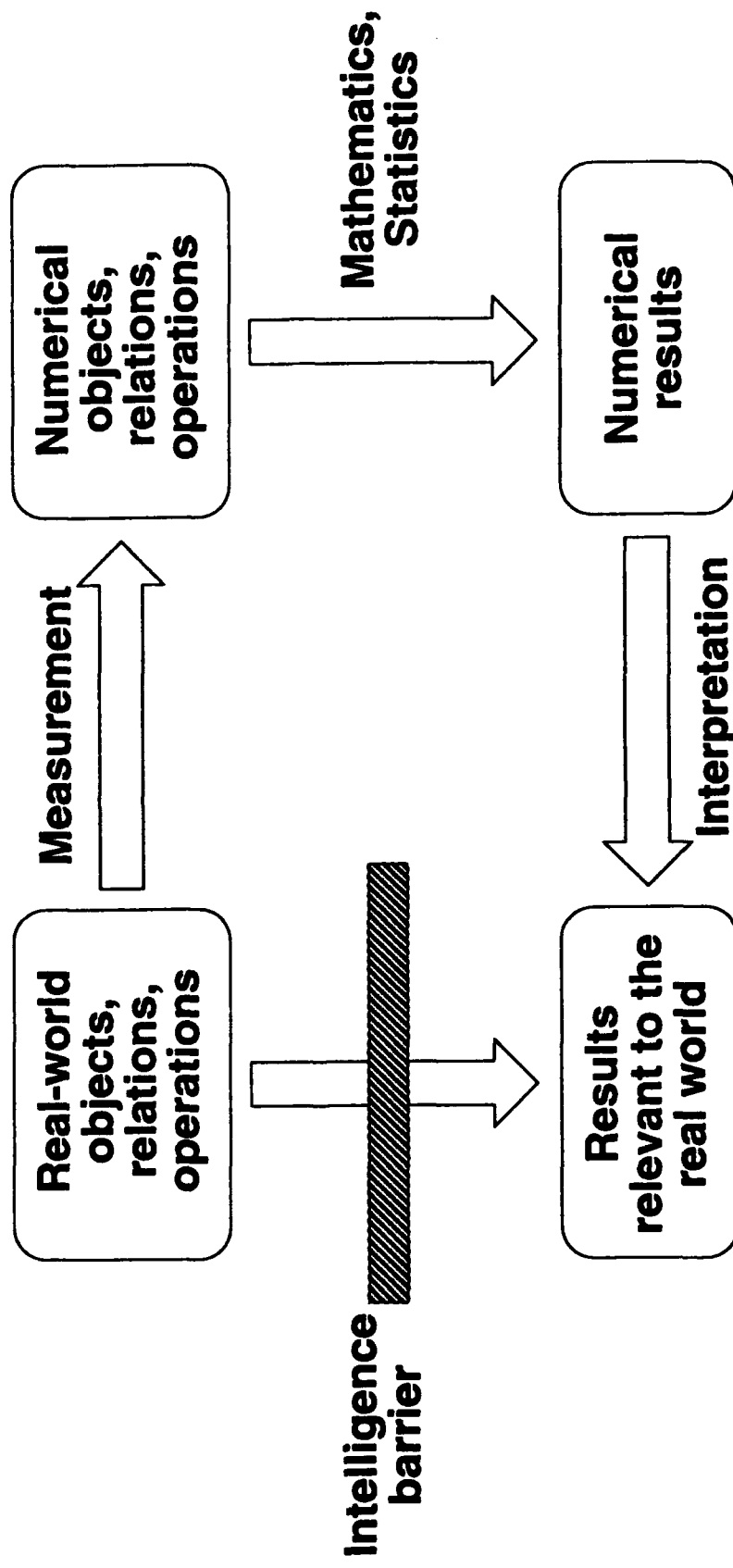
Metrics

Euclidean distance: $d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

“Manhattan” metric: $m(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$



Overcoming the Intelligence Barrier with Measurement



Definition

A relational system is defined as an ordered tuple $(S, rel_1, \dots, rel_n, op_1, \dots, op_m)$, where:

- S is a nonempty set of objects;**
- $rel_1, \dots, rel_n$ are k_i - ary relations on objects in S ;**
- $op_1, \dots, op_m$ are binary operations on objects in S .**

Definition

Let $A = (S_A, relA_1, \dots, relA_n, opA_1, \dots, opA_m)$ be a relational system of physical or empirical objects, and let $B = (S_B, relB_1, \dots, relB_n, opB_1, \dots, opB_m)$ be a relational system of formal objects (such as numbers). Let $\mu: S_A \rightarrow S_B$ be a measure. Then the triple (A, B, μ) is a *scale* if and only if

$$relA_j(a_{i_1}, \dots, a_{i_k}) \Leftrightarrow relB_j(\mu(a_{i_1}), \dots, \mu(a_{i_k}))$$

and

$$\mu(a opA_j b) = \mu(a) opB_j \mu(b)$$

for all values of i and j , and for all $a, b, a_{i_1}, \dots, a_{i_j} \in S_A$.

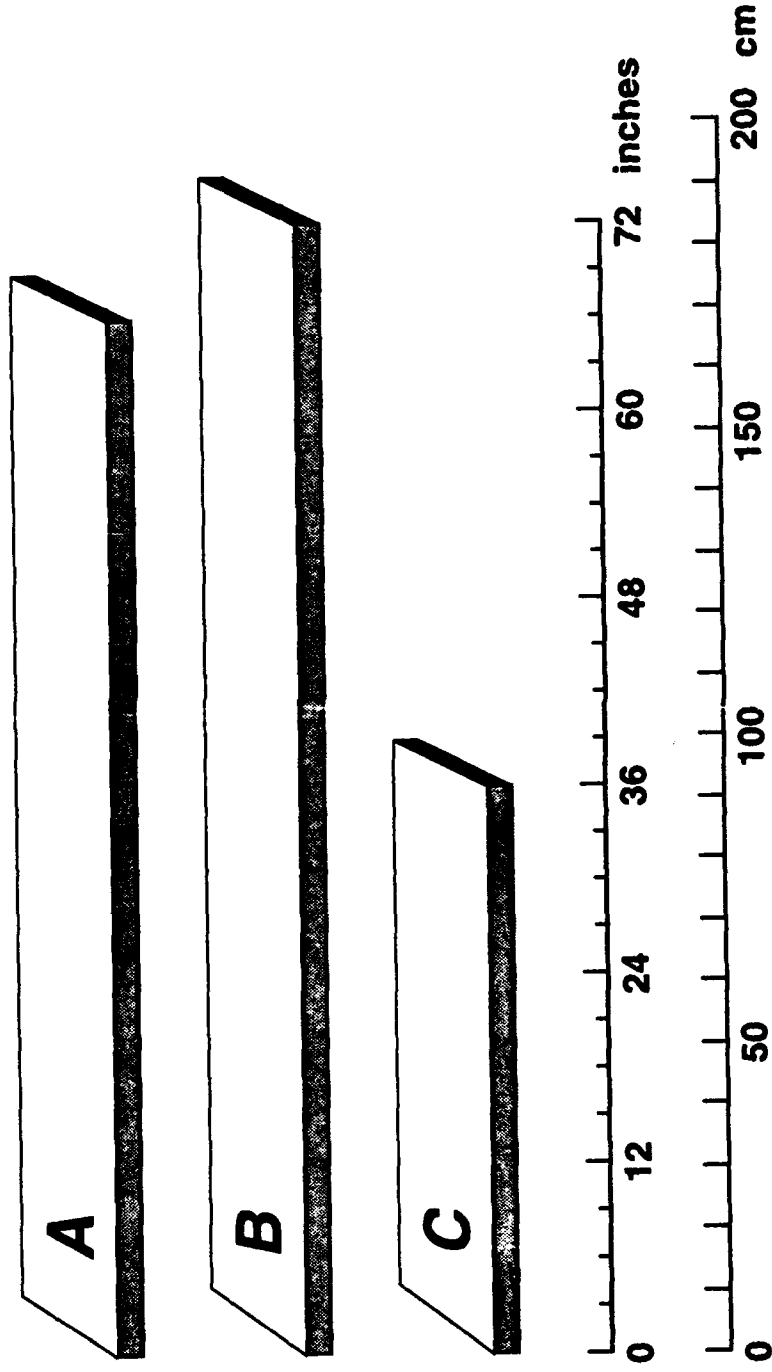
Definition

Let (A, B, μ) be a scale, where the set of objects in B is the set of real numbers. Let the notation $\mu(A)$ mean the set of all real numbers that are measures of some object in A (the range of μ). Then a mapping $t: \mu(A) \rightarrow B$ is defined to be an *admissible transformation* if and only if the triple $(A, B, t \circ \mu)$ is also a scale.

Definition

Let (A, B, μ) be a scale, where the set of objects in B is the set of real numbers. A statement about the measures $\mu(a)$ of objects in A is *meaningful* if and only if the truth value of that statement is unchanged after applying any admissible transformation to μ .

Meaningful Statements



Measurement Theory for Software Engineers: Figure 4

Discussion Question

For each of the following sets of objects, suggest a measure and scale for those objects, and identify the class in which the scale belongs (nominal, ordinal, interval, ratio, absolute).

- **Mass of physical objects**
- **Loudness of sounds**
- **Brightness of lights**
- **Human intelligence**
- **Beauty of the paintings in a museum**
- **Kelvin scale of temperature**
- **Size of a software system**
- **Productivity of different assembly line workers**

Discussion Question (continued)

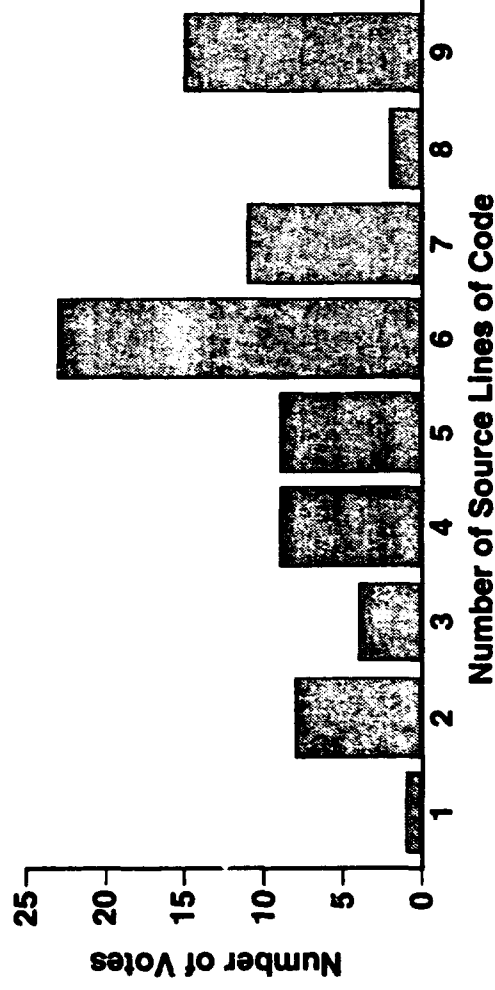
For each of the following sets of objects, suggest a measure and scale for those objects, and identify the class in which the scale belongs (nominal, ordinal, interval, ratio, absolute).

- **Productivity of different software engineers**
- **Cost of different models of automobiles**
- **Reliability of different models of automobiles**
- **Desirability of vacationing in each of the 50 states of the US**
- **Earthquake intensity**
- **Speed of different models of computer**
- **User-friendliness of word-processing or spreadsheet software**

Counting Lines of Code

```
#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));
}
```



Class Exercise: How Many Lines of Code?

```
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;
```

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 ☐	
<i>When a line or statement contains more than one type, classify it as the type with the highest precedence.</i>				Includes	Excludes
Order of precedence ->					
1 Executable		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			✓
Banners and nonblank spacers		6			✓
Blank (empty) comments		7			✓
Blank lines		8			✓
How produced		Definition ☐		Data array ☐	
1 Programmed			✓		
2 Generated with source code generators			✓		
3 Converted with automated translators			✓		
4 Copied or reused without change			✓		
5 Modified			✓		
Removed					✓

Staff-Hour Definition Checklist			
Definition Name:	<u>Total System Staff-Hours</u> <u>for Development</u>	Date:	<u>7/28/92</u>
		Originator:	
		Page:	<u>1 of 3</u>

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	✓		

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				
		✓		
		✓		
Uniqueness	Include	Exclude	Value Count	Array Count
Original	✓			
Duplicate		✓	✓	
Value not identified		✓		

Definition Checklist for Source Statement Counts

Definition name: _____ Date: _____

Originator: _____

Measurement unit:		Physical source lines ☐		
		Logical source lines ☐		
Statement type	Definition ☐ Data array ☐		Includes	Excludes
<i>When a line or statement contains more than one type, classify it as the type with the highest precedence.</i>				
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				
Origin	Definition ☐ Data array ☐		Includes	Excludes
1 New work: no prior existence				
2 Prior work: taken or adapted from				
3 A previous version, build, or release				
4 Commercial, of-the-shelf software (COTS), other than libraries				
5 Government furnished software (GFS), other than reuse libraries				
6 Another product				
7 A vendor-supplied language support library (unmodified)				
8 A vendor-supplied operating system or utility (unmodified)				
9 A local or modified language support library or operating system				
10 Other commercial library				
11 A reuse library (software designed for reuse)				
12 Other software component or library				
13				
14				
Usage	Definition ☐ Data array ☐		Includes	Excludes
1 In or as part of the primary product				
2 External to or in support of the primary product				
3				

Definition name: _____ _____				
Delivery 1 Delivered 2 Delivered as source 3 Delivered in compiled or executable form, but not as source 4 Not delivered 5 Under configuration control 6 Not under configuration control 7	Definition ☐	Data array ☐	Includes	Excludes
Functionality 1 Operative 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) 3 Functional (intentional dead code, reactivated for special purposes) 4 Nonfunctional (unintentionally present) 5 6	Definition ☐	Data array ☐	Includes	Excludes
Replications 1 Master source statements (originals) 2 Physical replicates of master statements, stored in the master code 3 Copies inserted, instantiated, or expanded when compiling or linking 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems 5	Definition ☐	Data array ☐	Includes	Excludes
Development status <i>Each statement has one and only one status, usually that of its parent unit.</i> 1 Estimated or planned 2 Designed 3 Coded 4 Unit tests completed 5 Integrated into components 6 Test readiness review completed 7 Software (CSCI) tests completed 8 System test completed 9 10 11	Definition ☐	Data array ☐	Includes	Excludes
Language <i>List each source language on a separate line.</i> 1 2 Job control languages 3 4 Assembly languages 5 6 Third generation languages 7 8 Fourth generation languages 9 10 Microcode 11	Definition ☐	Data array ☐	Includes	Excludes

Definition name: _____		Includes	Excludes
Listed elements are assigned to statement type →			
Clarifications (general)			
1 Nulls, continues, and no-ops			
2 Empty statements (e.g., ";;" and lone semicolons on separate lines			
3 Statements that instantiate generics			
4 Begin...end and {...} pairs used as executable statements			
5 Begin...end and {...} pairs that delimit (sub)program bodies			
6 Logical expressions used as test conditions			
7 Expression evaluations used as subprogram arguments			
8 End symbols that terminate executable statements			
9 End symbols that terminate declarations or (sub)program bodies			
10 Then, else, and otherwise symbols			
11 Elseif statements			
12 Keywords like procedure division, interface, and implementation			
13 Labels (branching destinations) on lines by themselves			
14			
15			
16			
Clarifications (language-specific)			
Ada			
1 End symbols that terminate declarations or (sub)program bodies			
2 Block statements (e.g., begin...end)			
3 With and use clauses			
4 When (the keyword preceding executable statements)			
5 Exception (the keyword, used as a frame header)			
6 Pragmas			
7			
8			
9			
Assembly			
1 Macro calls			
2 Macro expansions			
3			
4			
5			
6			
C and C++			
1 Null statement (e.g., ";" by itself to indicate an empty body)			
2 Expression statements (expressions terminated by semicolons)			
3 Expressions separated by semicolons, as in a "for" statement			
4 Block statements (e.g., {...} with no terminating semicolon)			
5 "{", "}", or ":" on a line by itself when part of a declaration			
6 "[" or "]" on line by itself when part of an executable statement			
7 Conditionally compiled statements (#if, #ifdef, #ifndef)			
8 Preprocessor statements other than #if, #ifdef, and #ifndef			
9			
10			
11			
12			

Definition name: _____		Includes	Excludes
Listed elements are assigned to statement type ↓			
CMS-2			
1	Keywords like SYS-PROC and SYS-DD		
2			
3			
4			
5			
6			
7			
8			
9			
COBOL			
1	"PROCEDURE DIVISION", "END DECLARATIVES", etc.		
2			
3			
4			
5			
6			
7			
8			
9			
FORTRAN			
1	END statements		
2	Format statements		
3	Entry statements		
4			
5			
6			
7			
8			
JOVIAL			
1			
2			
3			
4			
5			
6			
7			
8			
Pascal			
1	Executable statements not terminated by semicolons		
2	Keywords like INTERFACE and IMPLEMENTATION		
3	FORWARD declarations		
4			
5			
6			
7			
8			
9			

Staff-Hour Definition Checklist

Definition Name: _____ Date: _____
 _____ 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)			
Employment Class			
Reporting organization			
Full time			
Part time			
Contract			
Temporary employees			
Subcontractor working on task with reporting organization			
Subcontractor working on subcontracted task			
Consultants			
Labor Class			
Software management			
Level 1			
Level 2			
Level 3			
Higher			
Technical analysts & designers			
System engineer			
Software engineer/analyst			
Programmer			
Test personnel			
CSCI-to-CSCI integration			
IV&V			
Test & evaluation group (HW-SW)			
Software quality assurance			
Software configuration management			
Program librarian			
Database administrator			
Documentation/publications			
Training personnel			
Support staff			

Page: 2 of 3

Page: 2 of 3

[illegible]

Definition Name: _____

Page: 3 of 3[illegible]

Problem Count Definition Checklist

Software Product ID []		Page 1	
Definition Identifier: []		Definition Date []	
Attributes/Values	Definition []	Specification []	
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			
Criticality	Include	Exclude	Value Count Array Count
1st level (most critical)			
2nd level			
3rd level			
4th level			
5th level			
Value not identified			
Urgency	Include	Exclude	Value Count Array Count
1st (most urgent)			
2nd			
3rd			
4th			
Value not identified			

Problem Count Definition Checklist

Software Product ID [] Definition Identifier: []	Page 2 Definition Date []
Attributes/Values	Definition [] Specification []
Finding Activity	Include Exclude Value Count Array Count
Synthesis of	
Design	
Code	
Test procedure	
User publications	
Inspections of	
Requirements	
Preliminary design	
Detailed design	
Code	
Operational documentation	
Test procedures	
Formal reviews of	
Plans	
Requirements	
Preliminary design	
Critical design	
Test readiness	
Formal qualification	
Testing	
Planning	
Module (CSU)	
Component (CSC)	
Configuration item (CSCI)	
Integrate and test	
Independent verif. and valid.	
System	
Test and evaluate	
Acceptance	
Customer support	
Production/deployment	
Installation	
Operation	
Undetermined	
Value not identified	
Finding Mode	Include Exclude Value Count Array Count
Static (non-operational)	
Dynamic (operational)	
Value not identified	

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS None		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-93-EM-9			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute		6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office		
6c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213			7b. ADDRESS (city, state, and zip code) HQ ESC/ENS 5 Eglin Street Hanscom AFB, MA 01731-2116		
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office		8b. OFFICE SYMBOL (if applicable) ESC/AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003		
8c. ADDRESS (city, state, and zip code)) Carnegie Mellon University Pittsburgh PA 15213			10. SOURCE OF FUNDING NOS.		
			PROGRAM ELEMENT NO 63756E	PROJECT NO. N/A	TASK NO N/A
			WORK UNIT NO. N/A		
11. TITLE (Include Security Classification) Lecture Notes on Engineering Measruement for Software Engineers					
12. PERSONAL AUTHOR(S) Gary Ford					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM TO		14. DATE OF REPORT (year, month, day) April 1993	
15. PAGE COUNT 94 pp.					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)		
FIELD	GROUP	SUB. GR.	software engineering education software measurement		
			measurement engineering measurement		
			software metrics measurement theory		
19. ABSTRACT (continue on reverse if necessary and identify by block number)					
Measurement is a fundamental skill for engineers. To facilitate teaching software engineering mesaurement, materials are provided to support three lectures: introduction to engineering measurement, measurement theory, and software engineering measures. These materials include lecture notes suitable for class handouts and additional information for instructors—educational objectives, pedagogical considerations, suggestions for class projects, an annotated bibliography, and transparency masters for use in the delivery of the lectures.					
(please turn over)					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS ☒			21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution		
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Miller, Lt Col, USAF			22b. TELEPHONE NUMBER (include area code) (412) 268-7631		22c. OFFICE SYMBOL ESC/AVS (SEI)