Educational Materials
CMU/SEI-93-EM-9
April 1993

Lecture Notes on Engineering Measurement for Software Engineers

Gary Ford
Academic Education Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This document was prepared for the

SEI Joint Program Office
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

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 work was funded by the U.S. Department of Defense.

Table of Contents

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

Suggested Answers to Research Question 6

Name	Identification	Unit or Scale	Definition
James Clerk Maxwell	Scottish physicist 1831-1879	*maxwell:* 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	*Mohs scale:* mineral hardness scale	
Isaac Newton	English mathematician, physicist 1642-1727	*newton:* force	1 kilogram per second per second
Georg Simon Ohm	German physicist 1787-1854	*ohm:* resistance	resistance of a circuit in which a potential difference of one volt produces a current of one ampere
		mho: conductivity	ohm^{-1}
Hans Christian Ørsted	Danish physicist, chemist 1777-1851	*oersted:* 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	*pascal:* pressure	1 newton per square meter
Charles R. Richter	American seismologist 1900-1985	*Richter scale:* earthquake intensity scale	
Wilhelm Röntgen	German physicist 1845-1923	*roentgen:* 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	*tesla:* magnetic flux density	1 weber per square meter
Allesandro Volta	Italian physicist 1745-1827	*volt:* electromotive force; electrical potential difference	potential across one ohm when one ampere of current is flowing
James Watt	Scottish inventor 1736-1819	*watt:* power	one joule per second; one volt times one ampere
Wilhelm E. Weber	German physicist 1804-1891	*weber:* 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 ±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 ±5%, precision may be ±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 ±10°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 forms 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.; vanSchouwen, 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

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)