Appendix: Solutions to Selected Exercises

CHAPTER 2: THE BASICS OF MEASUREMENT

2. Answers to the questions:

   a. Nominal, ordinal, interval, ratio, and absolute are the five-scale types.

   b. The scale type for the complexity measure is ordinal, because there is a clear notion of order, but no meaningful notion of "equal difference" in complexity between, say, a trivial and simple module and a simple and moderate module. A meaningful measure of average for any ordinal scale data is the median. The mode is also meaningful, but the mean is not.

3. The answer is given in the following table:

Scale Type	Software Measure	Entity	Entity Type	Attribute
Nominal	Classification C, C++, Java, …	Compiler	Resource	Language
Ratio	Lines of code	Source code	Product	Size
Absolute	Number of defects found	Unit testing	Process	Defects found

14. A statement about measurement is meaningful if its truth value is invariant of the particular measurement scale being used.

The statement about Windows is meaningful (although probably not true) assuming that by size we mean something like the amount of code in the program. To justify its meaningfulness, we must consider how to measure this notion of size. We could use lines of code, thousands of lines of code, number of executable statements, number of characters, or number of bytes, for example. In each case, the statement's truth value is unchanged; that is, if a Windows program is about four times as big as the Linux program when we measure in lines of code, then it will also be about four times bigger when we measure in number of characters. We can say this because the notion of size is measurable on a ratio scale.

The rating scale for usability is (at best) ordinal, so there is no way to say that the difference in usability from 1 to 2 is the same as from 2 to 3. Therefore, if "average" refers to the mean, then the statement is not meaningful; changing to a different ordinal scale does not necessarily preserve the order of the mean values. However, if "average" refers to the median, then the statement is meaningful.

17. We must assume that the complexity ranking is ordinal, in the sense that the individual categories represent (from left to right) notions of increasing complexity.

    a. You can use the median of the M values. This choice is meaningful, because the measure M is an ordinal scale measure. Mode is also acceptable.

    b. Consider the following statement A: "The average complexity of modules in system X is greater than the average complexity of modules in system Y." We must show that A is not meaningful when "average" is interpreted as mean. To do this, we show that the truth value of the statement is not invariant of the particular measure used. Thus, consider the following two valid measures M and M':

	Trivial	Simple	Moderate	Complex	Very Complex	Incomprehensible
M	1	2	3	4	5	6
M'	1	2	3	4	5	10

    Suppose that the complexities of the modules in X are given by:

    x_1 trivial, x_2 and x_3 simple, x_4 moderate, x_5 incomprehensible

while the complexities of the modules in Y are given by:

$$y_1 \text{ simple}, \quad y_2, y_3, \text{ and } y_4 \text{ moderate}, \quad y_5, y_6, \text{ and } y_7 \text{ complex}.$$

Under M, the mean of the X values is 2.6, while the mean of the Y values is 3.1; so statement A is false under the measure M. However, under M', the mean of the X values is 3.6, while the mean of the Y values is 3.1. Thus, statement A is true under M'. From the definition of meaningfulness, it follows that the mean is not a meaningful average measure for this (ordinal scale) data.

c. Your answer might include a crude criterion based on some metric value, such as McCabe's cyclomatic complexity, v. For example, if $v(x) < 2$, we assign x as trivial; if $1 < v(x) < 5$, we assign x as simple, etc. Finally, if $v(x) > 100$, we assign x as incomprehensible. A more sophisticated criterion might be based on some measurable notion that closely matches our intuition about complexity. For example, the complexity of software modules is (intuitively) closely related to the difficulty of maintaining the modules. If maintenance data were available, such as mean time to repair (MTTR) faults for each module, then we could base our criterion on it. For example, if MTTR(x) < 30 min, then we assign x as trivial; if 30 min < MTTR(x) < 90 min, we assign x as simple, etc., until if MTTR(x) > 5 days, we assign x as incomprehensible.

22. The only way to report "number of bugs found" is to count number of bugs found. The answer to the second part depends very much on how you define program correctness. If you believe that a program is either correct or not, then correctness is measurable on a nominal scale with two values, such as (yes, no) or (1, 0). Or you may decide that correctness is measurable on a ratio scale, where the notion of zero incorrectness equals absolute correctness. (In this case, the number of bugs found could be a ratio-scale measure of program correctness.) In either of these cases, the number of bugs found cannot be an absolute scale measure for the attribute. Moreover, if "correctness" is taken to be synonymous with reliability, then counting bugs found may not be a measure of correctness at all, since we may not be finding the bugs that cause actual failures.

23. Answers to questions about the proposed *application domain (AD):*

    a. Clearly AD is a nominal measure as there is no notion of ordering; it is only a classification.

    b. The central tendency for ordinal measures is computed only as the mode, which is 1 (WWW browsers) for the given data.

    c. The statement "my compiler has three times the AD as your browser" is not meaningful.

24. Consider a proposed measure of *program adaptability:*

    a. There are lots of possibilities here. It could be time required to make a change, or better; yet, time required to make specific types of changes. Some structural measures can be OK, but then the empirical relations (below) are implied, and will not always be satisfied. The empirical relations should be of the nature of "it is more difficult to make some specific change to program A than program B".

    b. If the measure is time, then it is a ratio scale.

    c. It satisfies the representation condition if the measure is consistent with all empirical relations in *i*.

    d. Advantages and disadvantages vary.

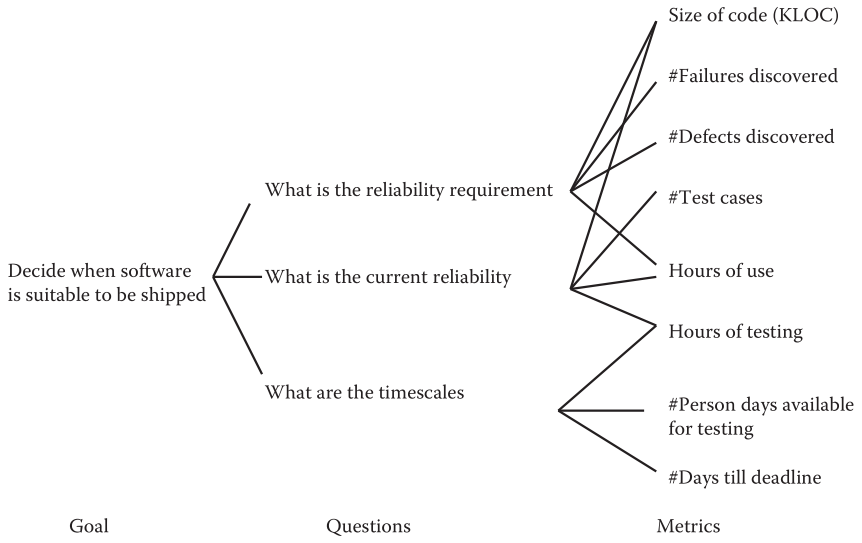
CHAPTER 3: A GOAL-BASED FRAMEWORK FOR SOFTWARE MEASUREMENT

5. GQM means Goal-Question-Metric, and it is based on the notion that measurement activities should always be preceded by identifying clear goals for them. To determine whether you have met a particular goal, you ask questions whose answers will tell you if the goal has been met. Then, you generate from each question the attributes you must measure in order to answer the questions.

   Goal-oriented measurement is common sense, but there are many situations where some measurement activity can be beneficial even though the goals are not clearly defined. This situation is especially true where a small number of metrics address several different goals, such as monitoring quality changes while predicting testing effort,

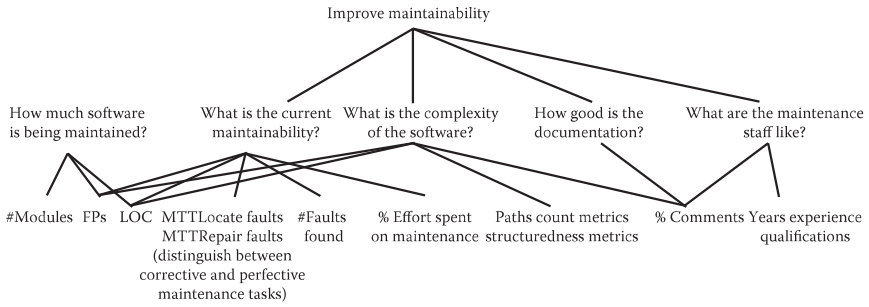
for example. Also, GQM introduces the problem of determining who sets the goals. High-level managers may be bad at identifying goals, or they may have goals for which no metrics can be practically computed at the engineering level. Because of this mismatch between goals and practicality, some people have proposed the need for bottom-up measurement, where the engineers collect metrics data that are useful and practical.

A typical GQM tree may resemble this one:



This tree depicts only direct measures. Questions such as "What is the reliability requirement?" are answered in terms of derived measures derived from the direct ones, and the tree can be extended to include both types. For example, the reliability requirement might be expressed as an average number of failures per 100 hours of use, or a number of defects per KLOC. Similarly, current reliability might be measured by number of defects per test case. It might also be useful if failures and defects were partitioned into two categories, "serious" (i.e.,, really affecting reliability) and "not serious," since many anomalies found during testing may be benign.

7. A typical GQM tree on improving maintainability of software may resemble the following figure:



CHAPTER 5: SOFTWARE METRICS DATA COLLECTION

1. An error is a mistake made by a human designer. A fault is the encoding of an error in the software. A failure is the manifestation of a fault during software execution.

4. Answers:

   a. Note first that only *project*-wide issues can be addressed because of the data's coarse granularity. Moreover, only goals and questions relating to project-wide comparisons are reasonable. Some possibilities are:

      i. General project-wide trends or comparisons in *productivity*, measured as the ratio of effort and lines of code. An example goal might be: "Improve project productivity," and a question might be: "What is current best, average, and worst productivity?" The well-known problems with this particular productivity measure should be cited here (e.g., lines of code do not really measure utility or even size of output; there are problems with the definition of the lines of code measure itself, etc.).

      ii. General project-wide trends or comparisons in *quality*, measured as the ratio of faults with lines of code. An example goal might be: "Improve product quality," and a question might be: "What is current best, average, and worst quality?" Note problems with the lines of code measure again, but the real

limitation is the use of faults. Total number of faults recorded may be more an indication of differences in recording or testing practice than of genuine quality. It does not reflect the quality of the end product.

iii. Provided you have data from a reasonable number of similar projects, it is possible to use the measures to improve project forecasting (i.e., effort estimation). You achieve this goal by using regression analysis on effort and lines of code data from the past projects.

iv. Provided you have data from a reasonable number of similar projects, you can use the measures for outlier analysis. For example, your goal may be to identify those projects having unusual productivity or quality values. However, all the limitations noted above still apply. To address this goal, you can use boxplots, two-dimensional scatter diagrams, etc.

b. You must collect data at the module level, rather than only at the project level. In particular, size, effort, and fault data should be available for each module. For each module, you must also identify the specific quality-assurance techniques that were used. (Ideally, you would know the extent of such use, measured by effort.) It would be helpful if the effort data per module could be partitioned into effort by activity, such as design, coding, testing, etc.

To get a more accurate perspective on whether a particular method has been effective, we need an external quality measure for each module. The number of faults discovered during testing is weak (and totally dependent on the level of testing and honesty of the tester). It would be better to obtain data on operational failures, where the failure can be traced back to individual modules.

To overcome the problems of lines of code as a poor size measure, it may be useful to collect other size data for each module. Static analysis tools or metrics packages offer inexpensive ways to collect structural metrics, which at least will indicate differences in structural complexity between modules (a difference not highlighted by lines of code). It may even be useful (although not as inexpensive) to use function point analysis.

5. Answers:

   a. There are several important aspects of dependability:

      i. *Safety* must be the main concern, since lives are endangered if the system malfunctions.

      ii. *Availability* is important, since it contributes to safety. Long or frequent "outages" will require personal supervision of patients by nursing staff, and drugs will have to be administered by hand.

      iii. *Reliability* is similarly important. Safety will be adversely affected by frequent failure, and frequent false alarms will cause unnecessary work for the medical staff, and may be to complacency, which could mean that a genuine warning is ignored. This and the following attributes may be considered to be of equal importance.

      iv. *Recoverability* is important since it contributes to high availability.

      v. *Maintainability* also contributes to high availability, and (in the case of removal of design faults) to growth in safety and reliability.

      vi. *Usability* is important from the point of view of the nursing staff, mainly because it contributes to safety by reducing the probability of human error, for example, in missing an indication that a patient requires attention.

   It is important to recognize that death or deterioration in a patient's condition is worse than those resulting only in inconvenience to staff or financial loss to the hospital. A case may be made for maintainability or recoverability being more important than availability or reliability. Security is of relatively minor concern, since the system is unlikely to be open to potential external interference. Features such as "fault-tolerance" assist reliability and safety, but are not themselves dependability attributes.

   b. Any of the following are acceptable:

      i. *Location*: Identity of the installation and piece of equipment on which the incident was observed.

      ii. *Timing(1):* When did the incident occur in real time?

iii. *Timing(2):* When did the incident occur in relation to system operational time, or software execution time?

iv. *Mode(1):* Type of symptoms observed.

v. *Mode(2):* Conditions of use of system.

vi. *Effect:* Type of consequence. This attribute will partly determine whether the incident affects safety, availability, or reliability.

vii. *Mechanism:* How did the incident come about?

viii. *Cause(1):* Types of trigger, such as physical hardware failure, operating conditions, malicious action, user error, erroneous report, "unexplained." This attribute will determine whether the incident is classified as relevant to usability or reliability.

ix. *Cause(2):* Type of source, such as physical hardware fault, unintentional design fault, intentional design fault, usability problem, "no cause found."

x. *Cause(3):* Identity of source (mode of failure and component in the case of physical failure, identity of fault in the case of design failure).

xi. *Severity(1):* Categorization of how serious the effect of the incident was. This will determine whether the incident affects safety.

xii. *Severity(2):* Cost to user (nursing staff or hospital) in terms of time lost, inconvenience, or extra effort incurred.

xiii. *Cost:* How much effort and other resources were expended by the vendor in order to diagnose and respond to the report of the incident?

xiv. *Count:* How many incidents occurred in given time interval?

c. These are examples of types of data that might be needed:

i. Records of design faults and design modifications. Operating time of each part of the system, and (ideally) execution.

ii. Time of each software module.

iii. The operational profile of the various parts of the system must also be recorded, since this will affect the levels of safety and reliability.

iv. To measure availability, recoverability, or maintainability, the time to restore service and the effort to diagnose and repair faults must be recorded.

v. To measure operating time and mode of operation accurately, it will be necessary to instrument the system, and write the data automatically to a file.

It is important to distinguish clearly between operating time and real time (calendar, or wall-clock, time), and to note that the operating time-base for the BED computer and for PAN computers (which execute many copies of the same code simultaneously) are different.

d. Five possible modes are the following:

i. PAN sends signal to operate drug pump when not required. Affects safety. *Effect:* Pump administers overdose. *Severity:* Critical/major (depending on the drug).

ii. BED software crashes. Affects availability, possibly safety. *Effect:* Loss of patient status information to nurse. *Severity:* Critical/major (depending on length of outage).

iii. PAN software does not detect that a sensor has failed. *Effect:* Relevant vital sign not monitored. *Severity:* Major.

iv. BED software indicates emergency although PAN has not reported deviation of vital signs. Affects reliability. *Effect:* Nurse visits patient unnecessarily. *Severity*: Minor.

v. "Wake up" signal not sent to junior doctor sufficiently often. Affects reliability, possibly safety. *Effect:* Delay in response of doctor to emergency. *Severity:* Major/minor.

6. Answers:

a. For each failure, it will be necessary to record:

i. *Time:* Both calendar time of the failure and total execution time of the software up to the failure (or alternatively, the

total execution time on all instances of the software during a given calendar time period) are required.

ii. *Location:* The particular instance of the software that failed, identified by the installation on which it is running. In this case, the control room will operate one (or more) instance(s) of the CAM software, and each taxi will operate an instance of the COT software.

iii. *Mode:* What symptoms were observed? That is, what was the system seen to do that was not part of its required functions?

iv. *Effect:* What were the consequences of the failure for the driver of the taxi and for the whole of the SCAMS operation?

v. *Severity*: How serious were these consequences? (This may be expressed either as a classification into major/minor/negligible, or in terms of the cost of lost business, or of the time that CAM or COT were out of action.)

vi. *Reporter:* Person making the report (taxi driver, control room operator, customer).

The following are five examples of software failure modes:

- *CAM software crashes*
  - *Time:* Date and time. Hours and minutes CAM was running until failure.
  - *Location:* Control room.
  - *Mode:* Loss of all screen displays, keyboard unresponsive, all COT systems lose contact with control room.
  - *Effect:* Loss of central control and guidance for all vehicles until CAM is to be rebooted and reestablishes contact with COT systems.
  - *Severity*: Major. Disruption of service for all drivers and passengers, with considerable inconvenience and loss of business.
  - *Reporter:* Control room operator.

- *COT software crashes*
  - *Time:* Date and time. Hours and minutes of all COT systems have been running until failure. (Alternatively, if that is not possible, total running hours on all COT systems on that date.)
  - *Location:* Number of the taxi whose COT failed, and where it was at the time. (The latter might be useful if the COT failure was in response to certain geographical data being input.)
  - *Mode:* Driver's display blank, driver's control console unresponsive, loss of communication of CAM with that vehicle.
  - *Effect:* Loss of central control and guidance for the affected vehicle until the driver could restart the COT.
  - *Severity:* Minor. Disruption of service for one driver and (possibly) passenger. Some loss of business (for instance, if driver could not pick up a passenger at the agreed place and time).
  - *Reporter*: Driver.
- *CAM sends incorrect directions to COT*
  - *Time:* Date and time. Hours and minutes CAM was running until failure.
  - *Mode:* Directions displayed on driver's screen are inconsistent with the actual position of the vehicle as observed by the driver.
  - *Effect*: Driver confused, and possibly gets lost.
  - *Severity:* Minor, unless the same CAM failure affects many vehicles.
  - *Reporter:* Driver or control room operator (depending on who first notices the incompatibility).
- *COT reports incorrect location to CAM*
  - *Time:* Date and time. Operating time of all COT systems.
  - *Mode:* Apparent discontinuity in vehicle movement as tracked by CAM.

- *Effect*: CAM cannot send correct directions to COT. Driver confused or lost.
- *Severity:* Minor. Affects one vehicle only.
- *Reporter:* Driver or control room operator.

- *Journey time is calculated by CAM to be shorter than it should be*
  - *Time:* Date and time. Hours and minutes CAM was running until failure.
  - *Mode:* ETA displayed by CAM is found to be inaccurate when compared to driver's actual reported journey time.
  - *Effect:* Driver arrives to pick up passenger later than arranged.
  - *Severity:* Minor, unless the inaccuracy affects many journeys.
  - *Reporter:* Driver.

b. There are several important dependability attributes.

Safety: The only critical failure mode is the failure of the silent alarm. Target: 1 failure per thousand demands (maximum).

Reliability: Targets will differ for different modes of failure:

i. For crash of COT software: 1 per 1000 operating hours.

ii. For crash of CAM software: 1 per 10,000 operating hours.

iii. For wrong directions: 1 per 5000 CAM operating hours, etc.

Recoverability: Target is to have the software recover after crash or error in an acceptable time. Different targets will apply to CAM and COT:

i. CAM: reload and resume service after crash within 1 min on average.

ii. COT: reload and resume service after crash within 10 s on average.

iii. Other targets may be set for recovery from lesser error conditions; for example, detection of CAM that it has given wrong directions, and alerting driver.

Availability: Target is to have the whole system up and running for an acceptable proportion of real time. Subsidiary targets may be set for individual subsystems; for example,

i. CAM: 99.8% up time.

ii. COT: 98% up time.

Maintainability: Target can be set for diagnosing and fixing a software fault.

c. Reliability, usability, safety, and security all relate to our expectation that certain types of events will *not* occur in operation. They can be measured by "the probability that the system will deliver a required service, under given conditions of use, for a given time interval, without relevant incident."

Reliability relates to departures of any kind from the required service. Usability relates to difficulties experienced by a human user attempting to operate the system, and roughly equates to "user-friendliness." Safety is concerned only with incidents that can result in death, injury, or other catastrophic damage. Security relates to incidents that result in unauthorized disclosure or alteration of sensitive information, or in denial of service to legitimate users by unauthorized persons.

It is therefore necessary to record all incidents, and measure their location, time of occurrence, the execution time over which they occur, mode, effect, and severity. Once again, it is necessary to record the execution time for CAM and COT software separately. The COT execution time needs to be totaled over all vehicles, and this will probably mean that only "failure count" data are available for COT, whereas "time to failure" data can be obtained fairly easily for CAM.

Once an incident has been shown to be a genuine failure and diagnosed as being due to a software fault, the identity of the fault responsible should be recorded. The type of fault can be used to establish to which attribute a given incident is relevant. The fault identity and cross-reference from each record of a failure in which it manifested itself can be used to extract "execution time up to first manifestation of each fault" (for CAM), or "count of faults manifest and total execution time in a given period" (for COT).

Measurement of maintainability requires the recording of the effort to diagnose and repair each fault. Recoverability requires records of time to restore service after each failure. A measure of availability can be derived from reliability and recoverability.

CHAPTER 6: ANALYZING SOFTWARE MEASUREMENT DATA

10. Answers:

    a. The standard quality measure in this context is defect density, calculated as number of defects divided by thousands of lines of code. The defect density for each system is given in the table below:

Module	Defect Density
A	0.88
B	0
C	190
D	0.7
E	0.46
F	0.57
G	0.92
H	0.4
I	1.125

    b. Box plots reveal that the defect metric has two outliers: one high (system C, with 95 defects) and one low (system B, with no defects). The lines of code metric has no outliers. The defect density metric has one outlier (over 15 times bigger than any other), namely system C at 19 defects/KLOC.

    c. Clearly, subsystem C is the problem area. Despite the fact that it is exceptionally small (just 4KLOC), it contains an abnormally high number of defects. You would be well advised to investigate this subsystem further to determine why. Is it exceptionally complex? Is it the part of the system most used? Has it the worst programmers or tools? There is also something unusual about subsystem B. Although large, this subsystem had no defects. Are there positive lessons to learn from it (for example, was it developed with the best programmers or tools?), or is it simply that this subsystem was not used?

d. A major weakness is that there is no notion of usage; subsystem C may be used much more than others, and system B may be never used. There are also no notions of severity of defects nor of complexity of subsystems. (KLOC is just a crude measure of size.) There is a blurring of the key distinction between fault and failure. Why should a failure always be due to a single defect in a single subsystem?

e. It would be extremely useful to get some measure of usage for each subsystem, since this would enable us to use the existing data to produce genuine reliability assessments of the different subsystems. Since system users are already logging failures as they occur, it should not be too difficult to get them to log the way they use the system. We should at least be able to get some estimate of the relative calendar time spent using the different subsystems. It would not be too difficult to insert some monitoring code into the system so that this can be done automatically. Another addition to the data-collection should be the measurement of actual time to locate and fix faults. This would reveal important maintainability information. Given the size of the system and the number of defects per year, maintainability is a major concern.

11. The outliers for CFP are modules R and P. The outliers for faults are P, Q, and R. So, the outliers for CFP and faults are more or less the same, whereas LOC has none. This result suggests that a high CFP is a more reliable indicator of a faulty module than high LOC. Thus, it would be a wise testing strategy to compute CFP for each module in a system and devote more time to testing, or redesigning, those that are outliers. Contrary to what many will say, there may be no point at all in redesigning modules R and P to lower their CFP, as the faults have already occurred!

12. The outliers of MOD and FD are the same, namely systems A, D, and L. Systems A and D are outliers of MOD because they have exceptionally low average module size, while system L is an outlier because it has exceptionally high average module size. Each of these three systems also had abnormally high fault density. This suggests that those systems in which average module size is either very high or very low are likely to be the most fault-prone. On the basis of this data, the outliers of MOD are excellent predictors of the high outliers of

FD. A reasonable quality assurance procedure would therefore be to restrict module sizes (on average) to be between 16 and 88 LOC. This procedure should avoid systems whose fault-proneness is explained by module size. A stricter quality procedure might focus on values between the upper and lower quartiles. In other words, we should restrict module sizes (on average) to be between 43 and 61 LOC. From the data, we can also conclude that size of a system alone (measured by KLOC) gives no indication of its fault proneness, nor of its likely average module size. For example, the only outlier for KLOC—the exceptionally large system R—does not have either an unusual average module size or an unusually high fault density.

CHAPTER 8: MEASURING INTERNAL PRODUCT ATTRIBUTES: SIZE

7. Function points are a measure of the functionality of a software system. The unadjusted function count UFC is derived from counting system inputs, outputs, enquiries, and files. A technical complexity factor, F, is then computed for the system, and the function point count is FP = UFC*F.

   The main applications of FPs are:

   a. Sizing for purposes of effort/cost estimation (provided that you have data about previous projects that relates the number of FPs in a system to the actual cost/effort).

   b. Sizing for purposes of normalization. Thus, FPs are used to compute quality density (defects/FP), productivity (person-months/FP), etc.

   Comparing FPs with LOC:

   a. Unlike LOC, FPs can be extracted early in software life-cycle (from requirements definition or specification) and so can be used in simple cost-estimation models where size is the key parameter.

   b. FPs, being a measure of functionality, are more closely related to utility than LOC.

   c. FPs are language-independent.

d. FPs can be used as a basis for contracts at the requirements phase.

However:

a. FPs are difficult to compute, and different people may count FPs differently.

b. Unlike LOC, FPs cannot be automatically extracted.

c. There is some empirical evidence to suggest that FPs are not very good for predicting effort. Empirical evidence also suggests that FPs are unnecessarily complex.

8. Answers:

a. Function points are supposed to measure the amount of functionality in a software "product," where product can mean any document from which the functional specification can be extracted: the code itself, the detailed design, or the specification. Function points are defined in a language-independent manner, so the number of function points should not depend on the particular product representation. Function points are also commonly interpreted as a measure of size.

Drawbacks:

i. The main drawback of function points is the difficulty in computing them. You must have at least a very detailed specification. This task is not easily automated or even repeatable. Different people will generally arrive at a different FP count for the same specification, although the existence of standards helps minimize the variance.

ii. The definition of function points was heavily influenced by the assumption that the number should be a good predictor of effort; in this sense, the function point measure is trying to capture more than just functionality. Thus, FPs are not very well-defined from the measurement theory perspective.

iii. FPs have been shown to be unnecessarily complicated. In particular, the TCF appears to add nothing in terms of measuring functionality, nor does it help to improve the predictive accuracy when FPs are used for effort prediction.

b. Most common answers will be:

   i. FPs can be used as the main input variable in an effort prediction system. Traditionally, LOC (or some similar code-based metric) has been used. The advantage of FPs is that they can be computed directly from the specification, so you need not predict LOC early in development.

   ii. Function points can be used in any application where you need to normalize by size. So, if you measure productivity traditionally by LOC/effort, it would be advantageous to use FPs instead of LOC, since FPs are more obviously related to the utility of output, and they are independent of the language used. You could also use FPs in this same equation to measure designer (or even specifier) productivity and not just coder productivity.

   iii. You could use FPs to measure defect density as defects/FP, rather than defects/LOC.

c. There is no single correct answer here. However, good answers should contain similar information.

   We identify the following items and their associated "complexity," weighting them as follows (using the table for UFC weighting factors):

External Inputs		
Coursework marks	Simple	3
Exam marks	Simple	3
Menu-selection: course choice	Simple	3
Menu-selection: operation choice	Simple	3
External Inquiries		
Average	Simple	3
Letter grade	Average	4
External Outputs		
List of student marks, etc.	Average	5
External Files		
None.		
Internal File		
Course file database: This contains the course list, student lists, and all known marks and grades.	Complex	10

Then UFC = 35

To compute the technical complexity factor (TCF), we consider the 14 listed factors in the Albrecht model. We assume that all but the following are irrelevant:

F6	Online data entry
F7	Operational ease
F9	Online update
F13	Multiple sites

Each of these is rated essential and hence get a weighting of 5.
Therefore, TCF = 0.65 + 0.01 * (4 * 5) = 0.85
Thus, FP = UFC * TCF = 30.

CHAPTER 9: MEASURING INTERNAL PRODUCT ATTRIBUTES: STRUCTURE

15. We look at each of the properties:

    a. Property 1—Nonnegativity: Does not hold. Any module or system with fan-out more than double the fan-in will have negative Cnew or CnewSyst.

    b. Property 2—Null value: Holds, as a very simple one module system can have no fan-in or fan-out.

    c. Property 3—Symmetry: Holds, as long as fan-in and fan-out links can be recognized.

    d. Property 4—Module monotonicity: Does not hold. A system with modules with very low Cnew values (very high fan-out) could cause CnewSyst to be less than the sum of two selected modules. This is a consequence of Property 1 being violated.

    e. Property 5—Disjoint module additivity: Does not hold. This is because individual Cnew and CnewSyst factors are computed differently. To compute CnewSyst, the total size of the system multiplies all fan-in and fan-out factors. The CnewSyst can vary dramatically from the sum of the Cnew values for individual modules, especially when the sizes of the systems vary greatly.

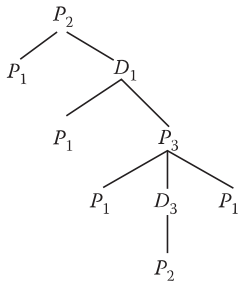
17. Coupling is a property of pairs of modules, while cohesion is a property of individual modules. Coupling between modules is the extent of interdependence between modules, whereas cohesion of a module is the extent to which the elements of the module have a common purpose, that is, are part of the same function.

    It is generally believed that if a design is made up of a number of related modules, then there should be (as far as possible) a low level of coupling; this way, errors made in any one module should affect a minimum number of others. Also, low coupling should help keep independent the implementation of the modules. On the other hand, high cohesion of each module may be desirable for conceptual simplicity and ease of testing. The lowest level of coupling can be achieved by having a single module for the whole system, but such a module will have very low cohesion. Analogously, we could ensure the highest level of cohesion of each module at the expense of very high coupling, namely where each individual statement corresponds to a module. Therefore, we have to find an optimal balance of low coupling and high cohesion.

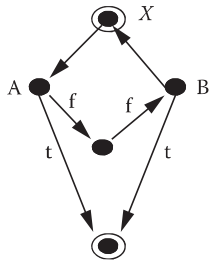
18. The software entity is source code (or the flowgraph representation of source code). The attribute is, strictly speaking, the number of linearly independent paths through the code, or the number of decisions plus one. The attribute is internal.

19. The statement coverage strategy is a form of white box testing. The tester must select inputs so that, when the program is executed, enough paths are executed for each statement of the program to lie on at least one path.

22. A procedure is *D*-structured, formally, if its decomposition tree (shown below) contains only primes of the form P_n, D_0, D_1, D_2, or D_3, which is true in this case:



23. The key thing to note about this algorithm is that its underlying flowgraph is the double-exit loop shown below.



In the strict interpretation of structured programming, only certain single-exit loops are allowed as building blocks. The two-exit loop is a prime structure that cannot be structured in terms of other such loops. Thus, in the strict sense, the algorithm is unstructured. But this example highlights the limitations of the strict view. Any attempt to rewrite the algorithm in "structured form" requires the introduction of new dummy variables which mar the simple and intuitive structure of the original algorithm. More liberal views of structured programming allow primes such as the two-exit loop, which is very natural in any control environment.

CHAPTER 10: MEASURING EXTERNAL PRODUCT ATTRIBUTES

2. The measure is quite useful for developers but is almost useless for users or potential purchasers. For developers who measure both LOC and faults found (in the code) in a consistent manner, the measure will indicate:

   a. Broad differences in quality among different modules, systems, teams

   b. Trends that can aid quality control efforts

   c. Potential troublespots in the system

   d. When the developers have reached the point of diminishing returns on testing

   However, the measure is of limited usefulness to the user, as it

   a. Gives no *real* indication of reliability (since faults may not be good predictor of failures, as shown in the Adams data)

b. Gives no indication at all of *usability*

c. Gives no indication of the severity of the faults

d. Cannot be used to compare products from different producers; the producers may count LOC differently, and they may have different definitions or classifications of faults

e. May say more about the rigor of the testing process (or of the tester) than about the quality of the code; clearly, the testing strategy influences how many faults are found

f. Invites abuse by programmers who may artificially increase the length of a program in order to be seen to be producing higher-quality code

g. Is irrelevant for assessing quality before coding has begun

h. Cannot be used comparatively with earlier products whose system size is not LOC

i. Does not take account of reused code

3. To compare projects with a given measure, it must be either normalized using size or type of project or independent of size/type of project. None of the measures here is normalized. Therefore, the only ones of value for comparative purposes are:

- Mean time to failure, for all projects at beta-test phase (a good measure of system reliability)
- Mean time to repair reported defects (a reasonable measure of maintainability)
- Maximum cyclomatic number (a very crude measure of system structuredness, and much less useful than the previous two measures)
- Average number of function points produced per month of programmer effort (a crude measure of programmer productivity; it would be dangerous to use this if the projects involved vastly different applications).

Each of the other measures, although useful for tracking and quality control purposes within a project, cannot be used sensibly for cross-project comparisons unless normalized.

- Total number of user-reported failures: For this to be used as a comparative measure of quality, it should be normalized against both system size and usage time. The former could be measured by total number of function points in the system (presumably already collected) or a simple measure like LOC. The latter would be much more difficult to measure, since it can be done only by the system users.
- Total number of defects found during system testing: At the very least, this measure should be normalized against size (as above). It may also need normalization against amount of testing.
- Total number of changes made during development: If this measure is normalized against size (preferably measured by function points), it may be a measure either of system volatility (if the changes are being suggested by customers) or quality (if the changes are being made as a result of discovering faults).
- Total project overspend/underspend: This measure should be normalized against actual expenditures to yield a measure of the accuracy of the cost predictions.

CHAPTER 11: SOFTWARE RELIABILITY: MEASUREMENT AND PREDICTION

8. Both prequential likelihood ratios actually converge at around $i = 10$ for this particular dataset. This result suggests that neither one is better than the other. (In fact, neither is particularly good.)

9. The totality of all possible inputs forms an input space. A fault can be thought of as a collection of points in the input space, each of which, when executed, results in output that is regarded as failed. (The decision to label an output failed will depend upon a comparison of what was required with what was produced.) Execution of the program involves the successive execution of a sequence of inputs. This sequence will be a random walk in the input space—random because the selection of future inputs is unpredictable and determined by the outside world. Such a trajectory through the input space will fall over the fault regions and so cause failures randomly.

   There is a further source of randomness (or uncertainty) when a fix is carried out. If the fix is successful, we are uncertain of the

magnitude of the effect of its removal on the unreliability. There will be a tendency for high rate faults to be encountered, and removed, earlier than low rate ones. In addition there is the possibility that a fix is not perfect. In this case, the effect on the program may be to increase the unreliability by an unknown amount.

10. The *u*-plot essentially detects consistent bias in predictions of reliability: the magnitude of the maximum departure from unit slope can be used to test whether the bias is statistically significant. The precise shape of the *u*-plot gives information about the nature of the errors. For example, a plot that is consistently above the line of unit slope tells us that the predictions are too optimistic.

    The prequential likelihood ratio allows us to compare the accuracies of prediction of model *A* with model *B* on the same data. If prequential likelihood ratio is consistently increasing as data vector increases in dimension, the model in numerator can be said to be more accurate than the model in the denominator. The prequential likelihood ratio is sensitive to all kinds of departure from the truth, and not just bias, but it is only a comparative analysis.

    The smoothed *u*-plot of previous predictions can be used to modify the current prediction as follows:

$$F_i^{\star}(t) = G_{i-1}\{F_i(t)\}$$

    where $F_i(t)$ is a raw estimate from a particular model of $P(T_i < t)$, G_{i-1} is the smoothed *u*-plot based on predictions of … t_{i-2}, t_{i-1}, and $F_i^{\star}(t)$ is the modified—recalibrated—prediction of $P(T_i < t)$.

11. The results clump into two groups: Six model predictions in one group, two in the other. There is reasonable agreement of the median predictions within each group, but big differences between groups. From the *u*-plot, all predictions are bad: 6 too optimistic, 2 too pessimistic. From the prequential likelihood ratio, the two pessimistic ones are not as bad as the six optimistic ones. None of the predictions can be trusted as they are; they should be recalibrated.

12. Answers:

    a. *Debugging data:* Use a reliability growth model, but you can get only quite modest reliability because of the law of diminishing

returns. You must also worry about the efficacy of fixes, since models tend to assume fixes are correct and do not introduce new faults; this is not a conservative assumption in the case of safety-critical systems.

b. *Failure-free working:* This is quite weak evidence of reliability. Roughly, we can expect there to be a 50:50 chance of working failure-free for a time t when we have seen a time t of failure-free working.

c. *Diversity:* There are experimental and theoretical reasons to doubt that the versions will fail independently. There is some evidence that this approach does deliver improvements over single version, but the improvement is hard to quantify. Evaluating a particular diverse system is equivalent to treating it as a black box, and thus the previous two paragraphs apply.

d. *Verification:* This technique addresses consistency with formal specification, but does not always address the issue of whether this is complete and accurate representation of the informal engineering requirements. There are practical difficulties with verification for any except small programs, because of resource constraints.

e. *Process:* There is very weak evidence to show a link between process and product.

14. Unless you know a lot about the faults you have removed, there is very little you can say about the improvements to reliability. It might be tempting to assume a proportional improvement in reliability, but on the basis of the Adams data, you could actually remove 95% of all faults and yet see no perceptible reliability improvements; this is because a very small proportion of faults cause almost all the common failures.