Sftare Metrics Overview

Measurement plays a critical rle in effective and efficient sftare develpment, as well as provides the scientific basis for software engineering that makes it a true engineering discipline. This course describes the software quality engineering metrics and models: quality planning, process improvement and quality control, inprocess quality management, product-engineering (design and code complexity), reliability estimation and projection, and analysis of customer satisfaction data.
We will cover in-process metrics for software testing, object-oriented metrics, availability metrics, in-process quality assessment, software project assessment, process improvement dos and don'ts, and measuring software process improvement.
The SW metrics are intended for use by software quality professionals; software project managers; software product managers; software development managers; software engineers; software product assurance personnel; and students in software engineering, management information systems, systems engineering, and quality engineering and management.
The course provides practical guidelines in the practice of quality engineering in software development. Although equations and formulas are involved, the focus is on the understanding and applications of the metrics and models rather than mathematical derivations.

software metric is a standard of measure of a degree to which a software system or process possesses some property. Even if a metric is not a measurement (metrics are functions, while measurements are the numbers obtained by the application of metrics), often the two terms are used as synonymous. Since quantitative measurements are essential in all sciences, there is a continuous effort by computer science practitioners and theoreticians to bring similar approaches to software development. The goal is obtaining objective, reproducible and quantifiable measurements, which may have numerous valuable applications in schedule and budget planning, cost estimation, quality assurance testing, software debugging, software performance optimization, and optimal personnel task assignments.

Acceptance and public opinion

Some software development practitioners point out that simplistic measurements can cause more harm than good. Others have noted that metrics have become an integral part of the software development process. Impact of measurement on programmers psychology have raised concerns for harmful effects to performance due to stress, performance anxiety, and attempts to cheat the metrics, while others find it to have positive impact on developers value towards their own work, and prevent them being undervalued. Some argue that the definition of many measurement methodologies are imprecise, and consequently it is often unclear how tools for computing them arrive at a particular result, while others argue that imperfect quantification is better than none (“You can’t control what you can't measure.”). Evidence shows that software metrics are being widely used by government agencies, the US military, NASA, IT consultants, academic institutions, and commercial and academic development estimation software.

Formally, we define measurement as a mapping from the empirical world to the formal, relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping in order to characterize an attribute.

Software metrics can be classified into three categories:

- product metrics,
- process metrics and
- project metrics.

Product metrics describe the characteristics of the product such as size, complexity, design features, performance, and quality level.
Process metrics can be used to improve software development and maintenance. Examples include the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fix process.
Project metrics describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity. Some metrics belong to multiple categories. For example, the inprocess quality metrics of a project are both process metrics and project metrics.
Software quality metrics are a *subset* of software metrics that focus on the quality aspects of the product, process, and project. In general, software quality metrics are more closely associated with process and product metrics than with project metrics. Nonetheless, the project parameters such as the number of developers and their skill levels, the schedule, the size, and the organization structure certainly affect the quality of the product. Software quality metrics can be divided further into end-product quality metrics and in-process quality metrics. The essence of software quality engineering is to investigate the relationships among in-process metrics, project characteristics, and end-product quality, and, based on the findings, to engineer improvements in both process and product quality. Moreover, we should view quality from the entire software life-cycle perspective and, in this regard, we should include metrics that measure the quality level of the maintenance process as another category of software quality metrics.
In this chapter we discuss **Management Metrics**:

- Size: Lines of Code (LOC*), Thousand Lines of Code (KLOC)
- Size: Function points, Feature Points
- Individual Effort: hours
- Task Completion Time: hours, days, weeks
- Project Effort: person-hours
- Project Duration: months
- Schedule: earned value
- Risk Projection: risk description, risk likelihood, risk impact

Quality of management has long been a concern within the software engineering community. The term ‘‘software crisis’’ was coined in the 1960s to refer to problems in developing software on time, within budget, and with the properties that the software was usable and actually used. The General Accounting Office reported in 1979 that of the government software development projects studied:

- more than 50% had cost overruns;
- more than 60% had schedule overruns;
- more than 45% of the delivered software could not be used;
- more than 29% of the software contracted for was never delivered;
- more than 19% of the delivered software had to be reworked

Since that report was released, the software engineering community has attempted to improve the software development process.
The quality of software development management tools has improved over the past 40 years. However,many of the same challenges, such as keeping software development projects on schedule and within budget, remain today. The Standish Group report 1995 found that, on average, approximately 16% of software projects were completed on time and within budget. In large companies the record was even worse: only 9% of the projects were completed on time and within budget. Moreover, the projects that were completed contained only approximately 42% of the originally proposed features and functions. The Center for Project Management in San Ramon, CA reported that 99% of commercial software products are not completed on time, within budget, or according to specifications, and that the average project is underestimated by 285%.
Capability maturity model (CMM) level-four compliance requires the development organization to collect metrics that measure the effectiveness of the development process, while CMM level five requires that the organization use the metrics continuously to improve its development process.
IEEE Standard 12207.0 include metrics that might be gathered during the software development process. Examples include software size and complexity, software units developed over time, milestone performance, and problem/change report status. Metrics are defined for the software development process and the software product but not for the quality of the project management. One can argue that in order to systematically go about improving the management of software projects, it is necessary to measure the quality of project management. Program-management tools have been developed to assist the program manager in estimating the cost and schedule of software programs. However, the estimation tools available assume consistent and high-quality program management. One of the earliest and most widely used software project cost-estimation models is COCOMO. The basic, intermediate, and detailed COCOMO models are based on the results of analyzing 63 software projects and applying regression analysis in order to predict software development cost as a function of software size and other factors. The intermediate and detailed COCOMO models take into account attributes of the software product, computer hardware, development personnel, and the project. Examples of project attributes include the use of software tools and the required development schedule. Intermediate cost estimates are based on the **estimated number of lines of code** (LOC) to be developed and then these estimates are adjusted by applying multipliers determined by rating the project with respect to the attributes. For example, a project completed under an accelerated schedule is estimated to cost more. However, COCOMO does not take into account the quality of project management.“
Poor management can increase software costs more rapidly than any other factor and despite this cost variation COCOMO *does not include a factor for management’s quality,* but instead provides estimates which **assume that the project will be well managed.**
If the quality of the software program management were measurable and available as input to costing and scheduling tools, the resulting estimates

could pin-point areas of software program management in which improvement needs to be made. Being able to measure the quality of management of software projects objectively allows development of more accurate cost models and would also provide a means for improving software project management through assessment, feedback, and correction. In this lecture, we introduce such a metric that is repeatable, termed the **quality management metric (QMM)**. We also discuss the informal and more formal validation of the metric. The QMM is computed from the quantitative answers to a structured set of inquiries, in a questionnaire consisting of two parts:
(1) a set of paired choices between statements that reflected possible management actions on a software program, and
(2) a set of questions requiring a yes, no, or not applicable answer.
The questionnaire was designed to eliminate essay-type answers and to minimize, as much as possible, subjective assessments. The questionnaire addresses four areas of software management considered to be the most important:

- requirements management,
- people management,
- risk management,
- and planning/estimation management.

We assume that, collectively, measures in the four areas can give an objective view of the quality of software management for a specific software development program. Thus, two programs scoring equally on product and process metrics can be further measured and compared on the basis of the quality of their management, thereby providing a more comprehensive look at a software program.
Estimations are the basis from which planning is performed on a program. Planning a software product development requires a frame of reference and an ability to measure against it. The program manager has three major measures with which to estimate the program: products, processes, and resources.
Product measures generally refer to volume, such as LOC. The measure can be the whole product or various elements, such as modules, components, or manuals. Measurement is accomplished by phase, such as the amount of code produced in the implementation phase or the LOC changed during unit testing. Measures of other product attributes might include system throughput, cyclomatic complexity, module coupling, and function points (FP).
Process measures quantify behavior, strategies, and execution of the process used to develop the product. One general category of process measures is event counts, such as the number of defects found in test, requirement changes, or milestones met. Another general category concerns time measures, such as cycle time: time to complete a project. In highly competitive markets, cycle time, or deployment, may be more important than reducing development costs.

In **procedural programming**, quality is measured by:

- *defects per thousand LOC* (KLOC),
- defects per function point,
- mean time to failure,

- and many other metrics and models.

Source lines of code (**SLOC**), also known as **lines of code** (**LOC**), is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code. SLOC is typically used to predict the amount of effort that will be required to develop a program, as well as to estimate programming productivity or maintainability once the software is produced.

Consider this snippet of C code as an example of the ambiguity encountered when determining SLOC:

```
for (i = 0; i < 100; i++) printf("hello"); /* How many lines of code is this? */
```

In this example we have:

- 1 Physical Line of Code (LOC)
- 2 Logical Lines of Code (LLOC) (for statement and printf statement)
- 1 comment line

Depending on the programmer and coding standards, the above "line of code" could be written on many separate lines:

```
/* Now how many lines of code is this? */
for (i = 0; i < 100; i++)
{
    printf("hello");
}
```

In this example we have:

- 5 Physical Lines of Code (LOC): is placing braces work to be estimated?
- 2 Logical Lines of Code (LLOC): what about all the work writing non-statement lines?
- 1 comment line: tools must account for all code and comments regardless of comment placement.

Even the "logical" and "physical" SLOC values can have a large number of varying definitions. Robert E. Park (while at the Software Engineering Institute) and others developed a framework for defining SLOC values, to enable people to carefully explain and define the SLOC measure used in a project. For example, most software systems reuse code, and determining which (if any) reused code to include is important when reporting a measure.

Origins

At the time that people began using SLOC as a metric, the most commonly used languages, such as FORTRAN and assembler, were line-oriented languages. These languages were developed at the time when punched cards were the main form of data entry for programming. One punched card usually represented one line of code. It was one discrete object that was easily counted. It was the visible output of the programmer so it made sense to managers to count lines of code as a measurement of a programmer's productivity, even referring to such as "card images". Today, the most commonly used computer languages allow a lot more leeway for formatting. Text lines are no longer limited to 80 or 96 columns, and one line of text no longer necessarily corresponds to one line of code.

Usage of SLOC measures

LOC measures are somewhat controversial, particularly in the way that they are sometimes misused. Experiments have repeatedly confirmed that effort is highly correlated with SLOC, that is, programs with larger SLOC values take more time to develop. Thus, SLOC can be very effective in estimating effort. However, functionality is less well correlated with SLOC: skilled developers may be able to develop the same functionality with far less code, so one program with fewer SLOC may exhibit more functionality than another similar program. In particular, SLOC is a poor productivity measure of individuals, since a developer can develop only a few lines and yet be far more productive in terms of functionality than a developer who ends up creating more lines (and generally spending more effort). Good developers may merge multiple code modules into a single module, improving the system yet appearing to have negative productivity because they remove code. Also, especially skilled developers tend to be assigned the most difficult tasks, and thus may sometimes appear less "productive" than other developers on a task by this measure. Furthermore, inexperienced developers often resort to code duplication, which is highly discouraged as it is more bug-prone and costly to maintain, but it results in higher SLOC

SLOC is particularly ineffective at comparing programs written in different languages unless adjustment factors are applied to normalize languages. Various computer languages balance brevity and clarity in different ways; as an extreme example, most assembly languages would require hundreds of lines of code to perform the same task as a few characters in APL. The following example shows a comparison of a "hello world" program written in C, and the same program written in COBOL - a language known for being particularly verbose.

C

```
# include <stdio.h>

int main() {
    printf("\nHello world\n");
}
```

COBOL

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID. HELLOWORLD.
000300
000400*
000500 ENVIRONMENT DIVISION.
000600 CONFIGURATION SECTION.
000700 SOURCE-COMPUTER. RM-COBOL.
000800 OBJECT-COMPUTER. RM-COBOL.
000900
001000 DATA DIVISION.
001100 FILE SECTION.
001200
100000 PROCEDURE DIVISION.
100100
100200 MAIN-LOGIC SECTION.
100300 BEGIN.
100400     DISPLAY " " LINE 1 POSITION 1 ERASE EOS.
```

```
100500     DISPLAY "Hello world!" LINE 15 POSITION 10.
100600     STOP RUN.
100700 MAIN-LOGIC-EXIT.
100800     EXIT.
```

Lines of code: 4 (excluding whitespace)	Lines of code: 17 (excluding whitespace)

Another increasingly common problem in comparing SLOC metrics is the difference between auto-generated and hand-written code. Modern software tools often have the capability to auto-generate enormous amounts of code with a few clicks of a mouse. For instance, graphical user interface builders automatically generate all the source code for a graphical control elements simply by dragging an icon onto a workspace. The work involved in creating this code cannot reasonably be compared to the work necessary to write a device driver, for instance. By the same token, a hand-coded custom GUI class could easily be more demanding than a simple device driver; hence the shortcoming of this metric.

There are several cost, schedule, and effort estimation models which use SLOC as an input parameter, including the widely used Constructive Cost Model (COCOMO) series of models by Barry Boehm et al., PRICE Systems True S and Galorath's SEER-SEM. While these models have shown good predictive power, they are only as good as the estimates (particularly the SLOC estimates) fed to them. Many have advocated the use of function points instead of SLOC as a measure of functionality, but since function points are highly correlated to SLOC (and cannot be automatically measured) this is not a universally held view.

The corresponding measure for defects per KLOC and defects per function point in **OO** is **defects per class**.

With regard to quality management, the OO design and complexity metrics can be used to flag the **classes with potential problems for special attention.** It appears that researchers have started focusing on the empirical validation of the proposed metrics and relating those metrics to managerial variables. This is certainly the right direction to strengthen the practical values of OO metrics.

Advantages

1. Scope for Automation of Counting: Since Line of Code is a physical entity; manual counting effort can be easily eliminated by automating the counting process. Small utilities may be developed for counting the LOC in a program. However, a logical code counting utility developed for a specific language cannot be used for other languages due to the syntactical and structural differences among languages. Physical LOC counters, however, have been produced which count dozens of languages.

2. An Intuitive Metric: Line of Code serves as an intuitive metric for measuring the size of software because it can be seen and the effect of it can be visualized. Function points are said to be more of an objective metric which cannot be imagined as being a physical entity, it exists only in the logical space. This way, LOC comes in handy to express the size of software among programmers with low levels of experience.
3. Ubiquitous Measure: LOC measures have been around since the earliest days of software. As such, it is arguable that more LOC data is available than any other size measure.

Disadvantages

1. Lack of Accountability: Lines of code measure suffers from some fundamental problems. Some think it isn't useful to measure the productivity of a project using only results from the coding phase, which usually accounts for only 30% to 35% of the overall effort.
2. Lack of Cohesion with Functionality: Though experiments have repeatedly confirmed that while effort is highly correlated with LOC, functionality is less well correlated with LOC. That is, skilled developers may be able to develop the same functionality with far less code, so one program with less LOC may exhibit more functionality than another similar program. In particular, LOC is a poor productivity measure of individuals, because a developer who develops only a few lines may still be more productive than a developer creating more lines of code - even more: some good refactoring like "extract method" to get rid of redundant code and keep it clean will mostly reduce the lines of code.
3. Adverse Impact on Estimation: Because of the fact presented under point #1, estimates based on lines of code can adversely go wrong, in all possibility.
4. Developer’s Experience: Implementation of a specific logic differs based on the level of experience of the developer. Hence, number of lines of code differs from person to person. An experienced developer may implement certain functionality in fewer lines of code than another developer of relatively less experience does, though they use the same language.
5. Difference in Languages: Consider two applications that provide the same functionality (screens, reports, databases). One of the applications is written in C++ and the other application written in a language like COBOL. The number of function points would be exactly the same, but aspects of the application would be different. The lines of code needed to develop the application would certainly not be the same. As a consequence, the amount of effort required to develop the application would be different (hours per function point). Unlike Lines of Code, the number of Function Points will remain constant.
6. Advent of GUI Tools: With the advent of GUI-based programming languages and tools such as Visual Basic, programmers can write relatively little code and achieve high levels of functionality. For example, instead of writing a program to create a window and draw a button, a user with a GUI tool can use drag-and-drop and other

mouse operations to place components on a workspace. Code that is automatically generated by a GUI tool is not usually taken into consideration when using LOC methods of measurement. This results in variation between languages; the same task that can be done in a single line of code (or no code at all) in one language may require several lines of code in another.

7. Problems with Multiple Languages: In today’s software scenario, software is often developed in more than one language. Very often, a number of languages are employed depending on the complexity and requirements. Tracking and reporting of productivity and defect rates poses a serious problem in this case since defects cannot be attributed to a particular language subsequent to integration of the system. Function Point stands out to be the best measure of size in this case.
8. Lack of Counting Standards: There is no standard definition of what a line of code is. Do comments count? Are data declarations included? What happens if a statement extends over several lines? – These are the questions that often arise. Though organizations like SEI and IEEE have published some guidelines in an attempt to standardize counting, it is difficult to put these into practice especially in the face of newer and newer languages being introduced every year.
9. Psychology: A programmer whose productivity is being measured in lines of code will have an incentive to write unnecessarily verbose code. The more management is focusing on lines of code, the more incentive the programmer has to expand his code with unneeded complexity. This is undesirable since increased complexity can lead to increased cost of maintenance and increased effort required for bug fixing.

Function point

A **function point** is a "unit of measurement" to express the amount of business functionality an information system (as a product) provides to a user. Function points are used to compute a functional size measurement (FSM) of software. The cost (in dollars or hours) of a single unit is calculated from past projects.
Function points were defined in 1979 in *Measuring Application Development Productivity* by Allan Albrecht at IBM. The functional user requirements of the software are identified and each one is categorized into one of five types: outputs, inquiries, inputs, internal files, and external interfaces. Once the function is identified and categorized into a type, it is then assessed for complexity and assigned a number of function points. Each of these functional user requirements maps to an end-user business function, such as a data entry for an Input or a user query for an Inquiry. This distinction is important because it tends to make the functions measured in function points map easily into user-oriented requirements, but it also tends to hide internal functions (e.g. algorithms), which also require resources to implement.

There is currently no ISO recognized FSM Method that includes algorithmic complexity in the sizing result. Recently there have been different approaches proposed to deal with this perceived weakness, implemented in several commercial software products. The variations of the Albrecht-based IFPUG method designed to make up for this (and other weaknesses) include:

- **Early and easy function points** – Adjusts for problem and data complexity with two questions that yield a somewhat subjective complexity measurement; simplifies measurement by eliminating the need to count data elements.
- **Engineering function points** – Elements (variable names) and operators (e.g., arithmetic, equality/inequality, Boolean) are counted. This variation highlights computational function. The intent is similar to that of the operator/operand-based Halstead complexity measures.
- **Bang measure** – Defines a function metric based on twelve primitive (simple) counts that affect or show Bang, defined as "the measure of true function to be delivered as perceived by the user." Bang measure may be helpful in evaluating a software unit's value in terms of how much useful function it provides, although there is little evidence in the literature of such application. The use of Bang measure could apply when re-engineering (either complete or piecewise) is being considered, as discussed in Maintenance of Operational Systems—An Overview.
- **Feature points** – Adds changes to improve applicability to systems with significant internal processing (e.g., operating systems, communications systems). This allows accounting for functions not readily perceivable by the user, but essential for proper operation.
- **Weighted Micro Function Points** – One of the newer models (2009) which adjusts function points using weights derived from program flow complexity, operand and operator vocabulary, object usage, and algorithmic intricacy.
- **Fast Function Points Analysis** (FFPA) – A similar system to IFPUG that was designed by Gartner as a way of calculating function points at a faster rate to deliver higher client benefit. It is presumably much faster than the traditional IFPUG method and only roughly 2% less accurate.

Benefits

The use of function points in favor of lines of code seek to address several additional issues:

- The risk of "inflation" of the created lines of code, and thus reducing the value of the measurement system, if developers are incentivized to be more productive. FP advocates refer to this as measuring the size of the solution instead of the size of the problem.
- Lines of Code (LOC) measures reward low level languages because more lines of code are needed to deliver a similar amount of functionality to a higher level language.
- LOC measures are not useful during early project phases where estimating the number of lines of code that will be delivered is challenging. However, Function

Points can be derived from requirements and therefore are useful in methods such as estimation by proxy.

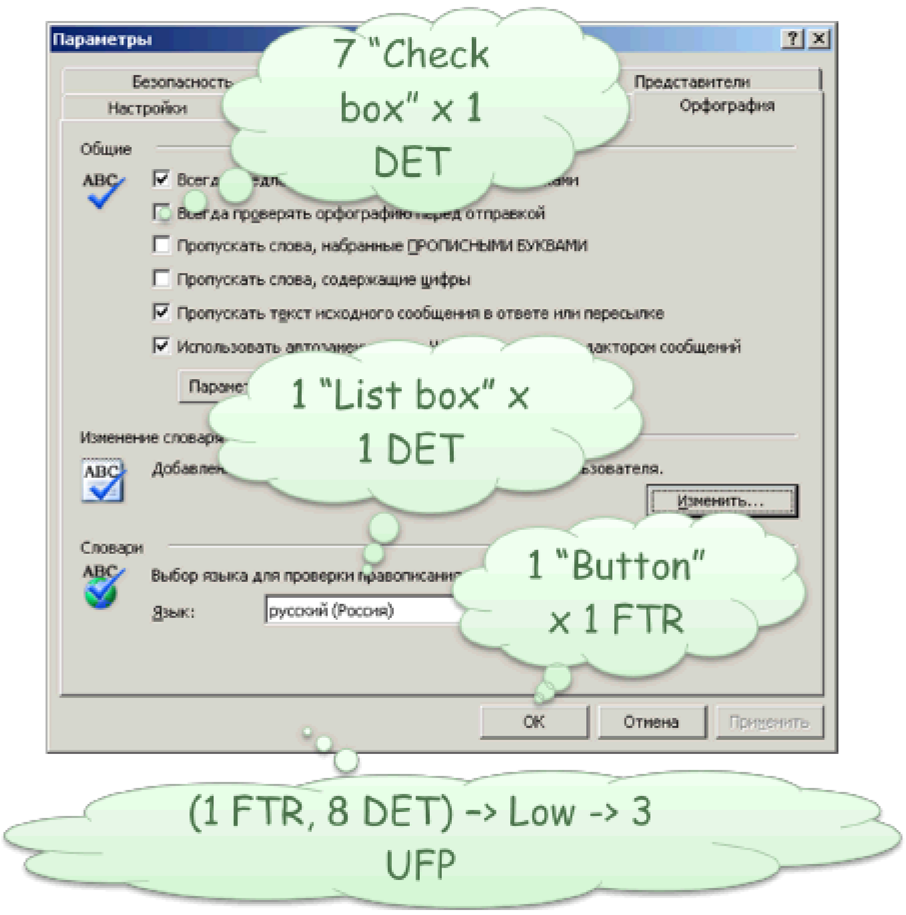
Criticism

Albrecht observed in his research that Function Points were highly correlated to lines of code, which has resulted in a questioning of the value of such a measure if a more objective measure, namely counting lines of code, is available. In addition, there have been multiple attempts to address perceived shortcomings with the measure by augmenting the counting regimen. Others have offered solutions to circumvent the challenges by developing alternative methods which create a proxy for the amount of functionality delivered.

Function Points

Albrecht’s effort estimation method was largely based on the notion of FPs. As their name suggests, FPs are intended to measure the amount of functionality in a system as described by a specification. We can compute FPs without forcing the specification to conform to the prescripts of a particular specification model or technique.
To compute the number of FPs we first compute an unadjusted function point count (UFC). To do this, we determine from some representation of the software the number of “items” of the following types:

- **External inputs**: Those items provided by the user that describe distinct application-oriented data (such as file names and menu selections). These items do not include inquiries, which are counted separately.
- **External outputs**: Those items provided to the user that generate distinct application-oriented data (such as reports and messages, rather than the individual components of these).
- **External inquiries**: Interactive inputs requiring a response.
- **External files:** Machine-readable interfaces to other systems.
- **Internal files**: Logical master files in the system.

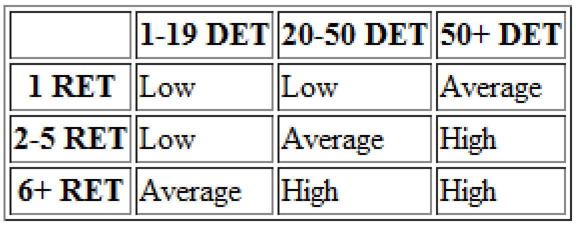

Параметры
Безопасность
Представители
Настройки
Орфография
Общие
Всегда проверять орфографию перед отправкой
Пропускать слова, набранные ПРОПИСНЫМИ БУКВАМИ
Пропускать слова, содержащие цифры
Пропускать текст исходного сообщения в ответе или пересылке
Изменение словаря
Изменить...
Словари
Выбор языка для проверки правописания
Язык: русский (Россия)
ОК
Отмена
Применить
7 "Check box" x 1 DET
1 "List box" x 1 DET
1 "Button" x 1 FTR
(1 FTR, 8 DET) -> Low -> 3 UFP

Counting function points associated with the data. Originally determined by the complexity of the data on the following parameters:

- **DET** (data element type) - unduplicated unique field data such as customer name - 1 DET; Address Client (code, country, region, district, city, street, house, building, apartment) - 9 DET's
- **RET** (record element type) - logical grouping of data, such as address, passport number.
  Estimates of functionality not aligned points depends on the complexity of the data, which is determined by the complexity of the matrix.

Counting function points associated with the transaction
Counting function points associated with the transaction - a fourth step analysis method function points.
Transaction - a closed elementary indivisible process, which is important for the user and translates product consistency from one state to another.
In the method distinguish these types of transactions :

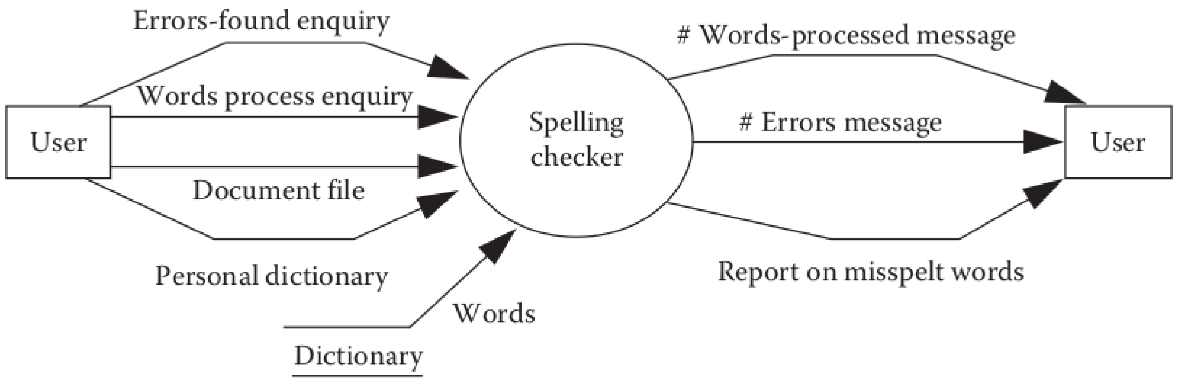
- EI (external inputs) - external input transactions elementary operation data or control information coming into the system from outside.
- EO (external outputs) - external outgoing transactions elementary operation to generate data or control information that go beyond the system. Suggest some logic processing computing or information from one or more ILF.
- EQ (external inquiries) - External requests elementary operation, which in response to an external request retrieves data or control information of ILF or EIF.



	1-19 DET	20-50 DET	50+ DET
1 RET	Low	Low	Average
2-5 RET	Low	Average	High
6+ RET	Average	High	High

EXAMPLE simple spelling checker.

Spell-checker spec: The checker accepts as input a document file and an optional personal dictionary file. The checker lists all words not contained in either of these files. The user can query the number of words processed and the number of spelling errors found at any stage during processing.



A = # external inputs = 2, B = # external outputs = 3, C = # inquiries = 2,
D = # external files = 2, and E = # internal files = 1

Illustration 1: simple spelling checker

Figure simple spelling checker describes a simple spelling checker. To compute the UFC from this description, we can identify the following items:

- The two **external inputs** are: document file-name, personal dictionary-name.
- The three **external outputs** are: misspelled word report, number-of-words-processed message, number-of-errors-so-far message.
- The **two external inquiries** are: words processed, errors so far.
- The two **external files** are: document file, personal dictionary.
- The one **internal file** is: dictionary.

Next, each item is assigned a subjective "complexity" rating on a three-point ordinal scale: simple, average, or complex.

Item	Simple	Weighting Factor Average	Complex
External inputs	3	4	6
External outputs	4	5	7
External inquiries	3	4	6
External files	7	10	15
Internal files	5	7	10

In theory, there are 15 different varieties of items (three levels of complexity for each of the five types), so we can compute the UFC by multiplying the number of items in a variety by the weight of the variety and summing over all 15:

$$\mathbf{UFC} = \sum_{i=1}^{15} (\text{ Number of items of variety } i \text{ }) \times (\text{ weight })$$

If we assume that the complexity for each item is average, then the UFC is

UFC = 4A + 5B + 4C + 10D + 7E = 58
If instead we learn that the dictionary file and the misspelled word report are considered complex, then

UFC = 4A + (5× 2 + 7× 1) + 4C + 10D + 10E = 63

To complete our computation of FPs, we calculate an adjusted function-point count, FP, by multiplying UFC by a technical complexity factor, TCF. This factor involves the 14 contributing factors:
Components of the Technical Complexity Factor
- F 1 Reliable backup and recovery
- F 3 Distributed functions
- F 5 Heavily used configuration
- F 7 Operational ease
- F 9 Complex interface
- F 11 Reusability
- F 13 Multiple sites
- F 2 Data communications
- F 4 Performance
- F 6 Online data entry
- F 8 Online update
- F 10 Complex processing
- F 12 Installation ease
- F 14 Facilitate change

Each component or subfactor is rated from 0 to 5, where 0 means the subfactor is irrelevant, 3 means it is average, and 5 means it is essential to the system being built. The following formula combines the 14 ratings into a final **technical complexity factor**:

$$\mathbf{TCF = 0.65 + 0.01 * \sum_{i=1}^{14} F_i}$$

This factor varies from 0.65 (if each F i is set to 0) to 1.35 (if each F i is set to 5). The final calculation of FPs multiplies the UFC by the technical complexity factor:

FP = UFC × TCF

EXAMPLE 3
To continue our FP computation for the spelling checker, we evaluate the technical complexity factor. After having read the specification, it seems reasonable to assume that F 3 , F 5 , F 9 , F 11 , F 12 , and F 13 are 0, that F 1 , F 2 , F 6 , F 7 , F 8 , and F 14 are 3, and that F 4 and F 10 are 5. Thus, we calculate the TCF as

TCF = 0.65 + 0.01(18 + 10) = 0.93
Since UFC is 63, then
FP = 63 × 0.93 = 59

FPs can form the basis for an effort estimate.

EXAMPLE 4
Suppose our historical database of project measurements reveals that it takes a developer an average of two person-days of effort to implement an FP. Then we may estimate the effort needed to complete the spelling checker as 118 days (i.e., 59 FPs multiplied by 2 days each).

The Common Software Measurement International Consortium (the "COSMIC"organization) http://www.cosmicon.com/
The latest definition of function point measurement is available from the International Function Point User Group (IFPUG). Information is available from http://www.ifpug.org/.
For a comprehensive review of the background and evolution of function point analysis, see the review paper by Christopher Lokan.
Lokan C.J., Function Points, Advances in Computers, 65, 297–344, 2005.
A good source for information about the use of COCOMO II object points is the COCOMO II website:
http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html

Cost and effort estimation
Managers provided the original motivation for deriving and using software measures. They wanted to be able to predict project costs during earlybphases in the software life cycle. As a result, numerous models for software cost and effort estimation have been proposed and used. Examples include Boehm's COCOMO II model (Boehm et al. 2000) and Albrecht's functionpoint model (Albrecht 1979). These and other models often share a common approach: effort is expressed as a (predefined) function of one or more variables (such as size of the product, capability of the developers, and level of reuse). Size is usually defined as (predicted) lines of code or number of function points (which may be derived from the product specification).

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.

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.

Person-month is politically correct synonym for **Man-month**.

It's mean amount of work performed by the average worker in one month.

So, if:

- project requires 12 persons-months of development time
- all team members do only pure development activity (i.e. they are telepaths and they don't need to spend time for communication with each other). *[note: this is not your case. In your case developers spend some time (5%) for communication.]*

than:

- 4 developers will spend 3 months for 12 persons-months project.

ProjectScopeInPersonsMonths / NumberOfDevelopers = NumberOfRequiredMonthsForProject

- 4 months required for 3 developers to finish 12 persons-months project.

ProjectScopeInPersonsMonths / NumberOfMonths = NumberOfRequiredDevelopersForProject

task completion time - a measure of the time it takes a user to perform a task (from start to finish). This is a typical metric in usability evaluation. Time to completion (TTC) is a calculated amount of time required for any particular task to be completed. Completion, is defined by the span from *conceptualization to fruition (delivery)*, and is not iterative. Similar to the metaphorical use of estimated time of arrival. TTC is commonly used when reporting on unmovable dates within a project time line. For example; a developer may report a TTC of 28 hours in regards to programming a particular application; although the application could perhaps be finished in 20 hours, the full allotted TTC will be fixed at 28 hours.

Earned_value_management
https://en.wikipedia.org/wiki/Earned_value_management

Risk projection, also called risk estimation, attempts to rate each risk in two ways—the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur. The project planner, along with other managers and technical staff, performs four risk projection activities:
(1) establish a scale that reflects the perceived likelihood of a risk,
(2) delineate the consequences of the risk,
(3) estimate the impact of the risk on the project and the product, and
(4) note the overall accuracy of the risk projection so that there will be no misunderstandings.

Developing a Risk Table

A risk table provides a project manager with a simple technique for risk projection .A project team begins by listing all risks (no matter how remote) in the first column of the table. This can be accomplished with the help of the risk item checklists Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk). The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. Individual team members are polled in round-robin fashion until their assessment of risk probability begins to converge.
Next, the impact of each risk is assessed. Each risk component is assessed and an impact category is determined. The categories for each of the four risk components—performance, support, cost, and schedule—are averaged to determine an overall impact value.
Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact. High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom. This accomplishes first-order risk prioritization.
The project manager studies the resultant sorted table and defines a cutoff line. The cutoff line (drawn horizontally at some point in the table) implies that only risks that lie above the line will be given further attention. Risks that fall below the line are re-evaluated to accomplish second-order prioritization. Risk impact and probability have a distinct influence on management concern. A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time. However, high-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow.

All risks that lie above the cutoff line must be managed. The column labeled RMMM contains a pointer into a Risk Mitigation, Monitoring and Management Plan or alternatively, a collection of risk information sheets developed for all risks that lie above the cutoff.
Risk probability can be determined by making individual estimates and then developing a single consensus value. Although that approach is workable, more sophisticated techniques for determining risk probability have been developed. Risk drivers can be assessed on a qualitative probability scale that has the following values: impossible, improbable, probable, and frequent. Mathematical probability can then be associated with each qualitative value (e.g., a probability of 0.7 to 1.0 implies a highly probable risk).

Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The nature of the risk indicates the problems that are likely if it occurs. For example, a poorly defined external interface to customer hardware (a technical risk) will preclude early design and testing and will likely lead to system integration problems late in a project. The scope of a risk combines the severity (just how serious is it?) with its overall distribution (how much of the project will be affected or how many customers are harmed?). Finally, the timing of a risk considers when and for how long the impact will be felt. In most cases, a project manager might want the “bad news” to occur as soon as possible, but in some cases, the longer the delay, the better.
Returning once more to the risk analysis approach proposed by the U.S. Air Force, the following steps are recommended to determine the overall consequences of a risk:
1. Determine the average probability of occurrence value for each risk component.
2. Determine the impact for each component based on the criteria
3. Complete the risk table and analyze the results as described in the preceding sections.
The overall risk exposure, RE, is determined using the following relationship:
RE = P x C
where P is the probability of occurrence for a risk, and C is the the cost to the project should the risk occur.
For example, assume that the software team defines a project risk in the following manner:

Risk identification. Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Risk probability. 80% (likely).

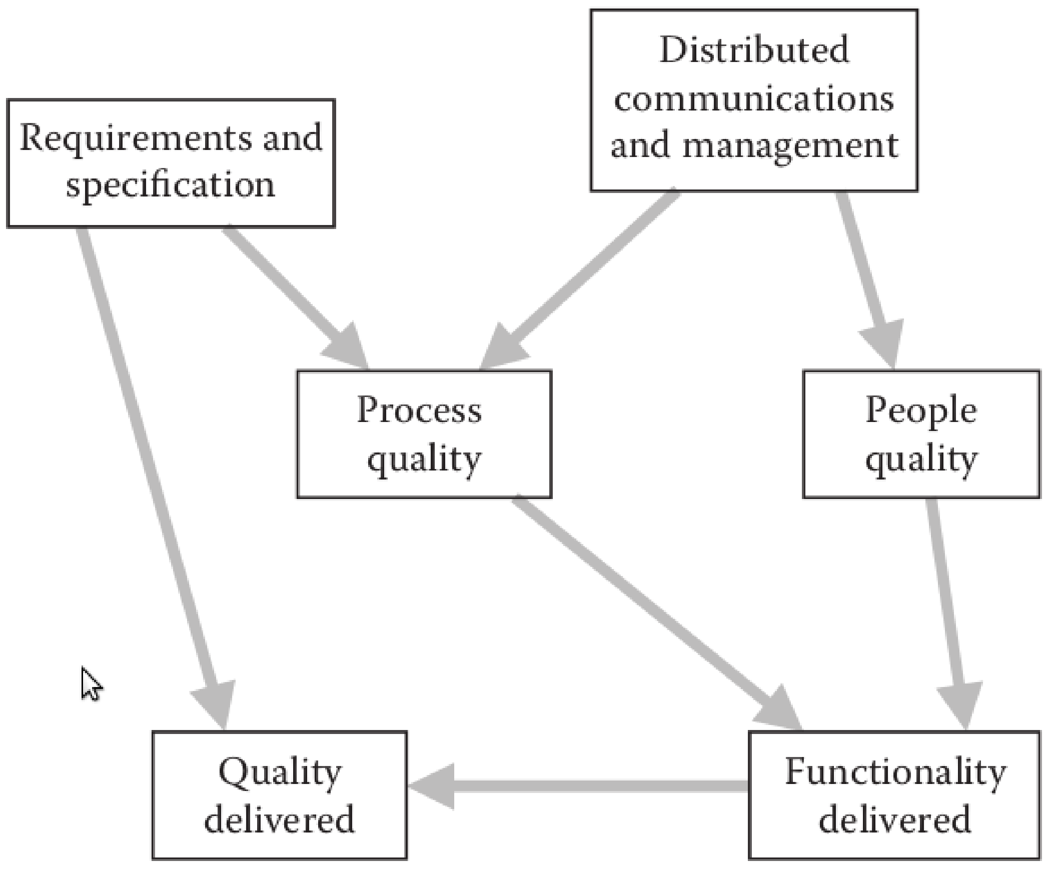
Risk impact. 60 reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is $14.00, the overall cost (impact) to develop the components would be
18 x 100 x 14 = $25,200.

Risk exposure. RE = 0.80 x 25,200 ~ $20,200.
Risk exposure can be computed for each risk in the risk table, once an estimate of the cost of the risk is made. The total risk exposure for all risks (above the cutoff in the risk table) can provide a means for adjusting the final cost estimate for a project. It can also be used

to predict the probable increase in staff resources required at various points during the project schedule.
The risk projection and analysis techniques are applied iteratively as the software project proceeds. The project team should revisit the risk table at regular intervals, re-evaluating each risk to determine when new circumstances cause its probability and impact to change. As a consequence of this activity, it may be necessary to add new risks to the table, remove some risks that are no longer relevant, and change the relative positions of still others.

Risk Assessment

At this point in the risk management process, we have established a set of triplets of the form:
[ri, li, xi]
where **ri is risk**, **li is the likelihood** (probability) of the risk, and **xi is the impact** of the risk. During risk assessment, we further examine the accuracy of the estimates that were made during risk projection, attempt to rank the risks that have been uncovered, and begin thinking about ways to control and/or avert risks that are likely to occur.
For assessment to be useful, a risk referent level must be defined. For most software projects, the risk components discussed earlier—performance, cost, support, and schedule—also represent risk referent levels. That is, there is a level for performance degradation, cost overrun, support difficulty, or schedule slippage (or any combination of the four) that will cause the project to be terminated. If a combination of risks create problems that cause one or more of these referent levels to be exceeded, work will stop. In the context of software risk analysis, a risk referent level has a single point, called the referent point or break point, at which the decision to proceed with the project or terminate it (problems are just too great) are equally weighted.
In reality, the referent level can rarely be represented as a smooth line on a graph. In most cases it is a region in which there are areas of uncertainty; that is, attempting to predict a management decision based on the combination of referent values is often impossible. Therefore, during risk assessment, we perform the following steps:
1. Define the risk referent levels for the project.
2. Attempt to develop a relationship between each (ri, li, xi) and each of the referent levels.
3. Predict the set of referent points that define a region of termination, bounded by a curve or areas of uncertainty.
4. Try to predict how compound combinations of risks will affect a referent level.


Distributed communications and management
Requirements and specification
Process quality
People quality
Quality delivered
Functionality delivered

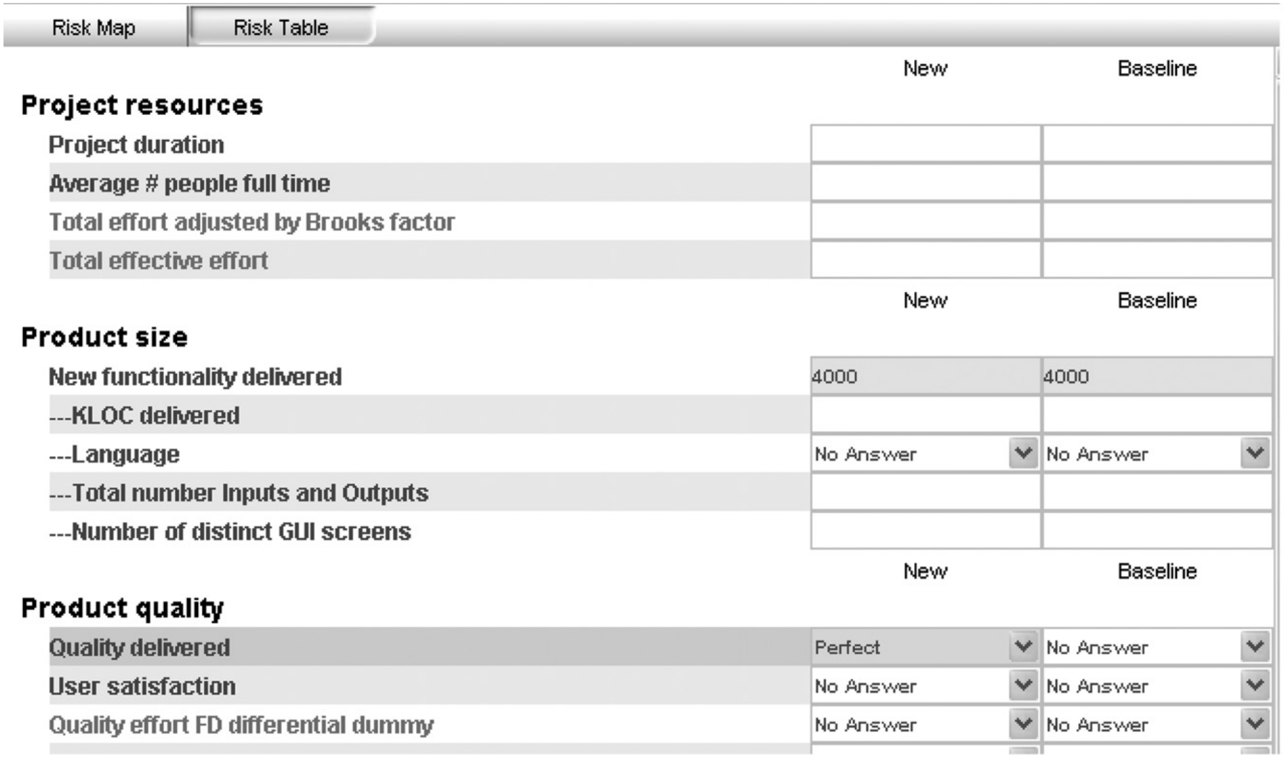
Full BN model for software project risk.

- **Distributed communications and management.** Contains variables that capture the nature and scale of the distributed aspects of the project and the extent to which these are well managed.
- **Requirements and specification.** Contains variables relating to the extent to which the project is likely to produce accurate and clear requirements and specifications.
- **Process quality**. Contains variables relating to the quality of the development processes used in the project.
- **People quality.** Contains variables relating to the quality of people working on the project.
- **Functionality delivered**. Contains all relevant variables relating to the amount of new functionality delivered on the project, including the effort assigned to the project.
- **Quality delivered.** Contains all relevant variables relating to both the final quality of the system delivered and the extent to which it provides user satisfaction (note the clear distinction between the two).

At its heart the model captures the classic trade-offs between:

- **Quality** (where we distinguish and model both user satisfaction—this is the extent to which the system meets the user's true requirements—and quality delivered—this is the extent to which the final system works well).
- **Effort** (represented by the average number of people full-time who work on the project).
- **Time** (represented by the project duration).

• **Functionality** (meaning functionality delivered).
So, for example, if you want a lot of functionality delivered with little effort in a short time then you should not expect high quality. If you need high quality then you will have to be more flexible on at least one of the other factors (i.e., use more effort, use more time, or deliver less functionality). What makes the model so powerful, when compared with traditional software cost models, is that we can enter observations anywhere in the model to perform not just predictions but also many types of trade-off analysis and risk assessment. So we can enter requirements for quality and functionality and let the model show us the distributions for effort and time. Alternatively, we can specify the effort and time we have available and let the model predict the distributions for quality and functionality delivered. Thus, the model can be used like a spreadsheet—we can test the effects of different assumptions.



	New	Baseline
Project resources		
Project duration		
Average # people full time		
Total effort adjusted by Brooks factor		
Total effective effort		
Product size	New	Baseline
New functionality delivered	4000	4000
---KLOC delivered		
---Language	No Answer	No Answer
---Total number Inputs and Outputs		
---Number of distinct GUI screens		
Product quality	New	Baseline
Quality delivered	Perfect	No Answer
User satisfaction	No Answer	No Answer
Quality effort FD differential dummy	No Answer	No Answer

Illustration 2: Two scenarios in risk table view.

To explain how this works we consider two scenarios called “New” and “Baseline”.Suppose the new project is to deliver a system of size 4000 function points (this is around 270 KLOC of Java, an estimate you can see for the node KLOC by entering the observation “java” for the question “language”). In the baseline scenario we enter no observations other than the one for functionality. We are going to compare the effect against this baseline of entering various observations into the new scenario. We start with the observations shown in Illustration *Two scenarios in risk table view*., that is, the only change from the baseline in the new project is to assert that the quality delivered should be “perfect.” Running the model produces the results shown in Figure 7.27 for the factors process and people quality, project
duration, and average number of people full time. First, note that the distributions for the latter factors have high variances (not unexpected given the minimal data entered) and that generally the new scenario will require a bit more effort for a bit longer. However, the factor process and people quality (which combines all the process and people factors) shows a very big difference from the baseline. The prediction already suggests that it will

be unlikely (a 14% chance) to deliver the system to the required level of quality unless the quality of staff is better than average.
Suppose, however, that we can only assume process and people quality is “**medium**.” Then the predictions for project duration and effort increase significantly. For example, the median value for project duration is up from 31 months in the baseline case to around 54 months
(a) Process and people quality,
(b) project duration (median 31, 39),
(c) average number of people full time (19, 23).) with full time staff increasing to 33.

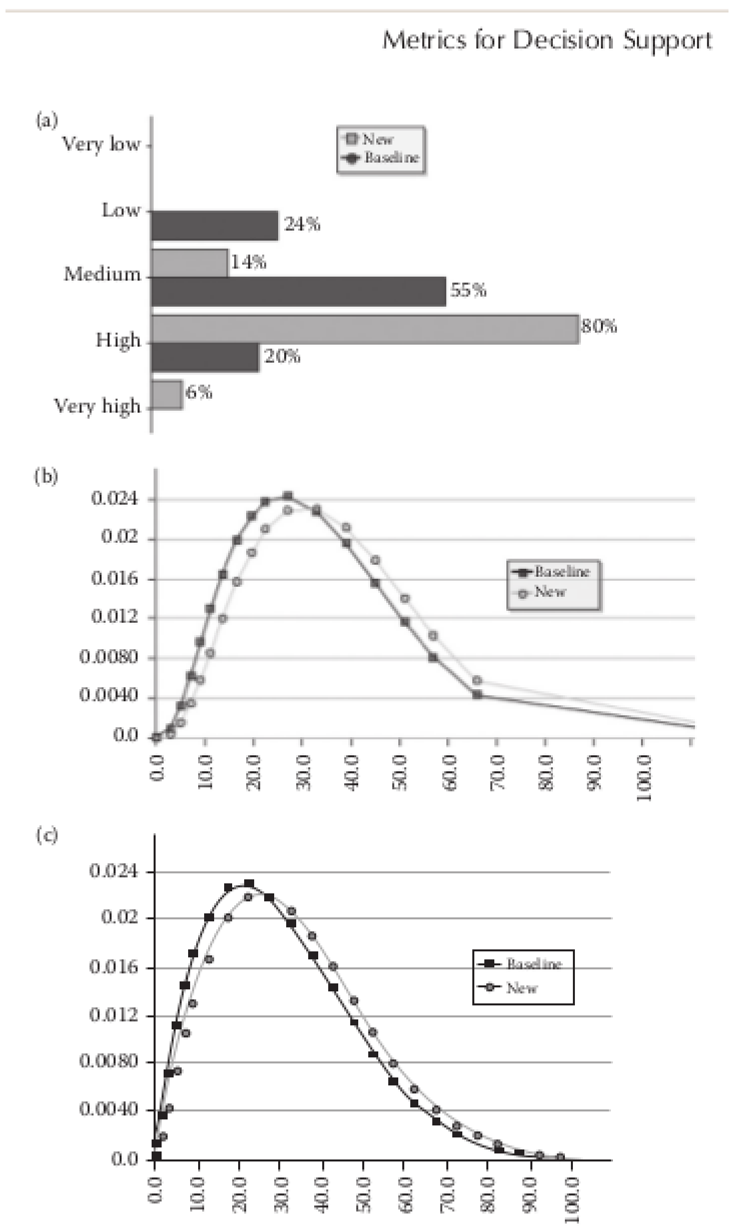


Illustration 3: Distributions when functionality delivered is set as “perfect” fornew project (compared with baseline). (a) Process and people quality, (b) projectduration (median 31, 39), (c) average number of people full time (19, 23).

Now, we withdraw the observation of process and people quality and suppose, as is typical in software projects, that we have a hard schedule deadline of 18 months in which to complete (i.e., a target that is significantly lower than the one the model predicts). With this observation we get the distributions shown in Figure 7.29 for process and people quality and average number of people full time. Now, not only do we need much higher quality people, we also need a lot more of them compared with the baseline. But typically, we will only have a fixed amount of effort. Suppose, for example that additionally we enter the observation that we have only 10 people full-time (so the project is really "under-resourced" compared with the predictions). Then the resulting distribution for process and people quality is shown in Figure 7.30.What we see now is that the probability of the overall process and people quality being "very high" (compared to the industry average) is 0.9966. Put a different way, if there is even a tiny chance that your processes and people are NOT among the best in the industry then this project will NOT meet its quality and resource constraints. In fact, if we know that the process and people quality is just "average" and now remove the observation "perfect" for quality delivered, then Figure 7.31 shows the likely quality to be delivered; it is very likely to be "abysmal" (with probability 0.69).

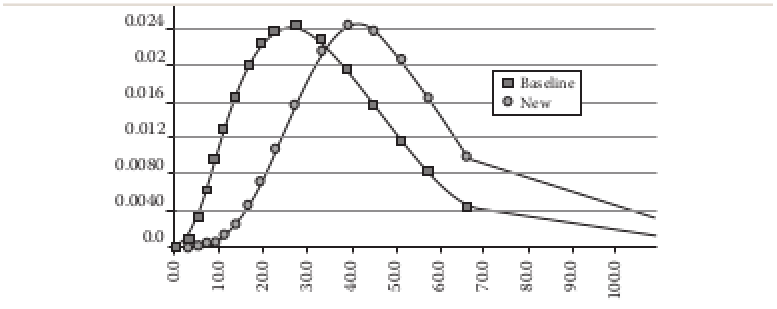


FIGURE 7.28 When staff quality is medium, project duration jumps to median value of 54 months.

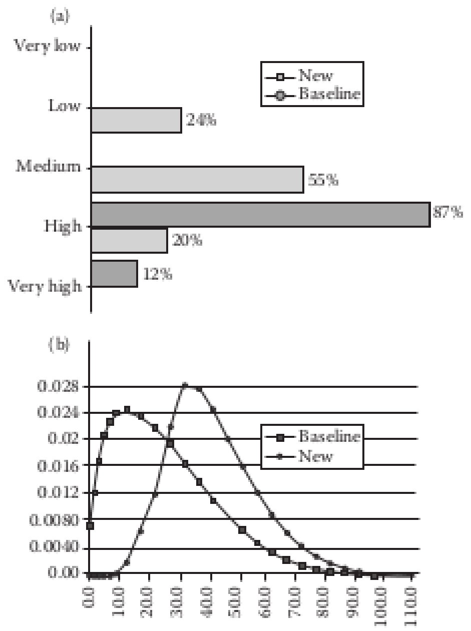


FIGURE 7.29 Project duration set to 18 months. (a) Process and people quality, (b) average number of people full time.

There have been many non causal models for software defect prediction and software resource prediction. Some of these have achieved very good accuracy and they provide us with an excellent empirical basis. However, in general these models are typically data-driven statistical models; they provide us with little insight when it comes to effective risk management and assessment.
What we have shown is that, by incorporating the empirical data with expert judgment, we are able to build causal Bayesian network models that enable us to address the kind of dynamic decision making that software professionals have to confront as a project develops.
The BN approach helps to identify, understand, and quantify the complex interrelationships (underlying even seemingly simple situations) and can help us make sense of how risks emerge, are connected and how we might represent our control and mitigation of them. By thinking about the causal relations between events we can investigate alternative explanations, weigh up the consequences of our actions and identify unintended or (un)desirable side effects. Above all else the BN approach quantifies the uncertainty associated with every prediction.
We are not suggesting that building a useful BN model from scratch is simple. It requires an analytical mindset to decompose the problem into "classes" of event and relationships that are granular enough to be meaningful, but not too detailed that they are overwhelming. The states of variables need to be carefully defined and probabilities need to be assigned that reflect our best knowledge. Fortunately, there are tools that help avoid much of the complexity of model building, and once built the tools provide dynamic and automated support for decision making. Also, so we have presented some pre-defined models that can be tailored for different organizations.

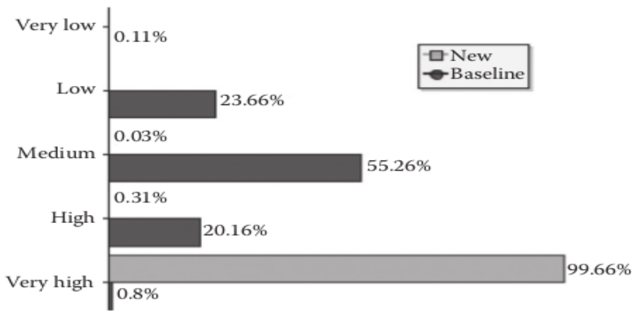


FIGURE 7.30 Project duration = 12 months, people = 10.

FURTHER READING

For an introduction and historical perspective of Bayes theorem and its applications:
McGrayne S.B., The Theory That Would Not Die, Yale University Press, CT, 2011.
Simpson E., Bayes at Bletchley Park, Significance, 7(2), 76–80, 2010.
For a comprehensive and not overtly mathematical overview of Bayesian networks and their applications and support, see:
Fenton N.E. and Neil M., Risk Assessment and Decision Analysis with Bayesian Networks, 2012, CRC Press, Boca Raton, FL, ISBN: 9781439809105, ISBN 10:1439809100, 2012.
There are also extensive resources available on the associated website: http://www.bayesianrisk.com/.
To understand the limitations of statistical modeling techniques and their tests of significance and p-values, see the following for a devastating critique of their widespread abuse across a range of empirical disciplines:

Ziliak S.T. and McCloskey D.N., The Cult of Statistical Significance: How the Standard Error Costs Us Jobs, Justice, and Lives, University of Michigan Press, Ann Arbor, USA, 2008.
Mathematically adept readers seeking more in depth understanding of the theoretical underpinnings of Bayesian networks and their associated algorithms should consider the following books:
Jensen F.V. and Nielsen T., Bayesian Networks and Decision Graphs, Springer-Verlag Inc, New York, 2007.
Madsen A.L., Bayesian Networks and Influence Diagrams, Springer-Verlag, New York, 2007.
Neapolitan R.E., Learning Bayesian Networks, Upper Saddle River Pearson Prentice Hall, 2004.