Unit II: Project Management: Process, Metrics, And Estimations & Risk

Software process and product metrics are quantitative measures that enable software people to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analyzed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred. Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved**.**

Software metrics refers to a broad range of measurements for computer software. Measurement can be applied to the software process with the intent of improving it on a continuous basis. Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. Finally, measurement can be used by software engineers to help assess the quality of technical work products and to assist in tactical decision making as a project proceeds.

If you don't measure, judgement can be based only on subjective evaluation. With measurement, trends (either good or bad) can be spotted, better estimates can be made and true improvement can be accomplished over time.

There are four reasons for measuring software processes, products, and resources: to characterize, to evaluate, to predict, or to improve. We characterize to gain understanding of processes, products, resources, and environments, and to establish baselines for comparisons with future assessments. We evaluate to determine status with respect to plans. Measures are the sensors that let us know when our projects and processes are drifting off track, so that we can bring them back under control. We also evaluate to assess achievement of quality goals and to assess the impacts of technology and process improvements on products and processes

MEASURES, METRICS, AND INDICATORS

Within the software engineering context, a measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. Measurement is the act of determining a measure. The IEEE Standard Glossary of Software Engineering Terms [IEE93] defines metric as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."

When a single data point has been collected (e.g., the number of errors uncovered in the review of a single module), a measure has been established. Measurement occurs as the result of the collection of one or more data points (e.g., a number of module reviews are investigated to collect measures of the number of errors for each).

A software metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per person- hour expended on reviews. A software engineer collects measures and develops metrics so that indicators will be obtained. An indicator is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself [RAG95]. An indicator provides

insight that enables the project manager or software engineers to adjust the process, the project, or the process to make things better.

For example, four software teams are working on a large software project. Each team must conduct design reviews but is allowed to select the type of review that it will use. Upon examination of the metric, errors found per person-hour expended, the project manager notices that the two teams using more formal review methods exhibit an errors found per person-hour expended that is 40 percent higher than the other teams. Assuming all other parameters equal, this provides the project manager with an indicator that formal review methods may provide a higher return on time investment than another, less formal review approach. She may decide to suggest that all teams use the more formal approach. The metric provides the manager with insight. And insight leads to informed decision making.

METRICS IN THE PROCESS AND PROJECT DOMAINS

Metrics should be collected so that process and product indicators can be ascertained. Process indicators enable a software engineering organization to gain insight into the efficacy of an existing process (i.e., the paradigm, software engineering tasks, work products, and milestones). They enable managers and practitioners to assess what works and what doesn't. Process metrics are collected across all projects and over long periods of time. Their intent is to provide indicators that lead to long-term software process improvement. Project indicators enable a software project manager to (1) assess the status of an ongoing project, (2) track potential risks, (3) uncover problem areas before they go "critical," (4) adjust work flow or tasks, and (5) evaluate the project team's ability to control quality of software work products. In some cases, the same software metrics can be used to determine project and then process indicators. In fact, measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process improvement. For this reason, many of the same metrics are used in both the process and project domain.

1. Process Metrics and Software Process Improvement

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement. But before we discuss software metrics and their impact on software process improvement, it is important to note that process is only one of a number of "controllable factors in improving software quality and organizational performance

Referring to Figure 4.1, process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance. The skill and motivation of people has been shown [BOE81] to be the single most influential factor in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods) that populate the process also has an impact. In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., CASE tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication). We measure the efficacy of a software process indirectly. That is, we derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include

measures of errors uncovered before release of the software, defects delivered to and reported by end-users, work products delivered (productivity), human effort expended, calendar time expended, schedule conformance, and other measures. We also derive process metrics by measuring the characteristics of specific software engineering tasks. For example, we might measure the effort and time spent

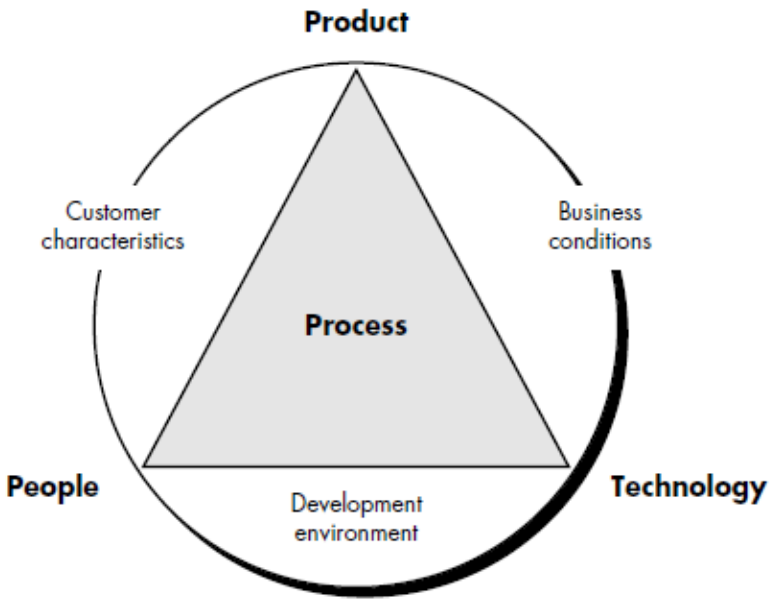


FIGURE 4.1 Determinants for software quality and organizational effectiveness (adapted from [PAU94])

Grady [GRA92] argues that there are "private and public" uses for different types of process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis, these data should be private to the individual and serve as an indicator for the individual only. Examples of private metrics include defect rates (by individual), defect rates (by module), and errors found during development.

Humphrey recognizes that software process improvement can and should begin at the individual level. Private process data can serve as an important driver as the individual software engineer works to improve.

Some process metrics are private to the software project team but public to all team members. Examples include defects reported for major software functions (that have been developed by a number of practitioners), errors found during formal technical reviews, and lines of code or function points per module and function. These data are reviewed by the team to uncover indicators that can improve team performance.

Public metrics generally assimilate information that originally was private to individuals and teams. Project level defect rates (absolutely not attributed to an individual), effort, calendar times, and related data are collected and evaluated in an attempt to uncover indicators that can improve organizational process performance.

Software process metrics can provide significant benefit as an organization works to improve its overall level of process maturity. However, like all metrics, these can be misused, creating

more problems than they solve. Grady [GRA92] suggests a "software metrics etiquette" that is appropriate for both managers and practitioners as they institute a process metrics program:

• Use common sense and organizational sensitivity when interpreting metrics data.

• Provide regular feedback to the individuals and teams who collect measures and metrics.

• Don't use metrics to appraise individuals.

• Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.

• Never use metrics to threaten individuals or teams.

• Metrics data that indicate a problem area should not be considered "negative." These data are merely an indicator for process improvement.

• Don't obsess on a single metric to the exclusion of other important metrics.

As an organization becomes more comfortable with the collection and use of process metrics, the derivation of simple indicators gives way to a more rigorous approach called statistical software process improvement (SSPI). In essence, SSPI uses software failure analysis to collect information about all errors and defects3 encountered as an application, system, or product is developed and used. Failure analysis works in the following manner:

1. All errors and defects are categorized by origin (e.g., flaw in specification, flaw in logic, nonconformance to standards).

2. The cost to correct each error and defect is recorded.

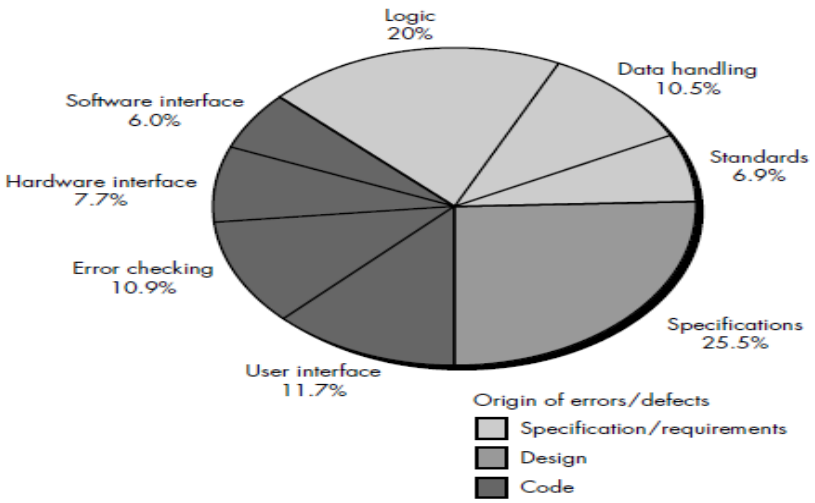
3. The number of errors and defects in each category is counted and ranked in descending order.

4. The overall cost of errors and defects in each category is computed.

5. Resultant data are analyzed to uncover the categories that result in highest cost to the organization.

6. Plans are developed to modify the process with the intent of eliminating (or reducing the frequency of) the class of errors and defects that is most costly.

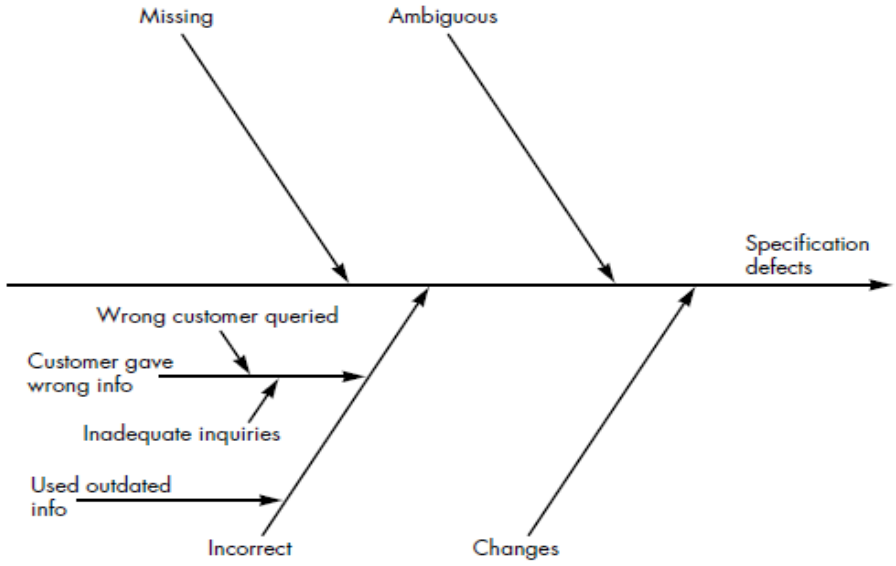
FIGURE 4.2
Causes of defects and their origin for four software projects [GRA94]



Following steps 1 and 2, a simple defect distribution can be developed (Figure 4.2) [GRA94]. For the pie-chart noted in the figure, eight causes of defects and their origin (indicated by shading) are shown. Grady suggests the development of a fishbone diagram [GRA92] to help in diagnosing the data represented in the frequency diagram. Referring to Figure 4.3, the spine of the diagram (the central line) represents the quality factor under consideration (in this case specification defects that account for 25 percent of the total). Each of the ribs (diagonal lines) connecting to the spine indicate potential causes for the quality problem (e.g., missing requirements, ambiguous specification, incorrect requirements, changed requirements). The spine and ribs notation is then added to each of the major ribs of the diagram to expand upon the cause noted. Expansion is shown only for the incorrect cause in Figure 4.3.

The collection of process metrics is the driver for the creation of the fishbone diagram. A completed fishbone diagram can be analyzed to derive indicators that will enable a software organization to modify its process to reduce the frequency of errors and defects.

FIGURE 4.3
A fishbone diagram (adapted from [GRA92])



2. Project Metrics

Software process metrics are used for strategic purposes. Software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work flow and technical activities.

The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of pages of documentation, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from specification into design, technical metrics are collected to assess design quality and to provide indicators that will influence the approach taken to code generation and testing.

The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost. Another model of software project metrics [HET93] suggests that every project should measure:

• Inputs—measures of the resources (e.g., people, environment) required to do the work.

• Outputs—measures of the deliverables or work products created during the software engineering process.

• Results—measures that indicate the effectiveness of the deliverables.

In actuality, this model can be applied to both process and project. In the project context, the model can be applied recursively as each framework activity occurs. Therefore the output from one activity becomes input to the next. Results metrics can be used to provide an indication of the usefulness of work products as they flow from one framework activity (or task) to the next.

SOFTWARE MEASUREMENT

Measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects). Software metrics can be categorized similarly. Direct measures of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported

over some set period of time. Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities" The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance. However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly.

Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure 4.4, can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project. Referring to the table entry (Figure 4.4) for project alpha: 12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000. It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project alpha indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released, and 29 defects

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

FIGURE 4.4
Size-oriented metrics

were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha. In order to develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

• Errors per KLOC (thousand lines of code).

• Defects4 per KLOC.

• $ per LOC.

• Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

• Errors per person-month.

• LOC per person-month.

• $ per page of documentation.

Size-oriented metrics are not universally accepted as the best way to measure the process of software development [JON86]. Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that they penalize well-designed but shorter programs, that they cannot easily accommodate nonprocedural languages, and that their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

4.3.2 Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since ‘functionality’ cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht [ALB79], who suggested a measure called the function point. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity. Function points are computed [IFP94] by completing the table shown in Figure 4.5. Five information domain characteristics are determined and counts are provided in the appropriate table location. Information domain values are defined in the following manner:

Number of user inputs. Each user input that provides distinct applicationoriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

Number of user outputs. Each user output that provides applicationoriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

Number of user inquiries. An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.

Number of files. Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces. All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

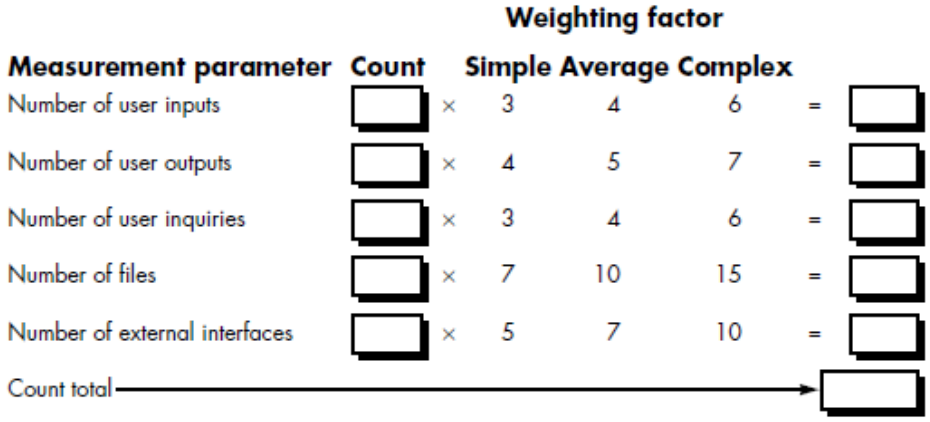
Once these data have been collected, a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective.

To compute function points (FP), the following relationship is used:

FP = count total * [0.65 + 0.01 * Σ(Fi)]

where count total is the sum of all FP entries obtained from Figure 4.5.

FIGURE 4.5 Computing function points



The *Fi* (*i* = 1 to 14) are "complexity adjustment values" based on responses to the following questions:

1. Does the system require reliable backup and recovery?

2. Are data communications required?

3. Are there distributed processing functions?

4. Is performance critical?

5. Will the system run in an existing, heavily utilized operational environment?

6. Does the system require on-line data entry?

7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?

8. Are the master files updated on-line?

9. Are the inputs, outputs, files, or inquiries complex?

10. Is the internal processing complex?

11. Is the code designed to be reusable?

12. Are conversion and installation included in the design?

13. Is the system designed for multiple installations in different organizations?

14. Is the application designed to facilitate change and ease of use by the user?

Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). The constant values in Equation (4-1) and the weighting factors that are applied to information domain counts are determined empirically.

Once function points have been calculated, they are used in a manner analogous to LOC as a way to normalize measures for software productivity, quality, and other attributes:

• Errors per FP.

• Defects per FP.

• $ per FP.

• Pages of documentation per FP.

• FP per person-month.

Extended Function Point Metrics

The function point measure was originally designed to be applied to business information systems applications. To accommodate these applications, the data dimension (the information domain values discussed previously) was emphasized to the exclusion of the functional and behavioral (control) dimensions. For this reason, the function point measure was inadequate for many engineering and embedded systems (which emphasize function and control). A number of extensions to the basic function point measure have been proposed to remedy this situation. A function point extension called feature points [JON91], is a superset of the function point measure that can be applied to systems and engineering software applications.

The feature point measure accommodates applications in which algorithmic complexity is high. Real-time, process control and embedded software applications tend to have high algorithmic complexity and are therefore amenable to the feature point. To compute the feature point, information domain values are again counted and weighted as described in Section 4.3.2. In addition, the feature point metric counts a new software characteristic—*algorithms.* An *algorithm* is defined as "a bounded computational problem that is included within a specific computer program". Inverting a matrix, decoding a bit string, or handling an interrupt are all examples of algorithms. Another function point extension for real-time systems and engineered products has been developed by Boeing. The Boeing approach integrates the data dimension of software with the functional and control dimensions to provide a function-oriented measure amenable to applications that emphasize function and control capabilities. Called the *3D function point*, characteristics of all three software dimensions are "counted, quantified, and

transformed" into a measure that provides an indication of the functionality delivered by the software.The *data dimension* is evaluated in much the same way. Counts of retained data (the internal program data structure; e.g., files) and external data (inputs, outputs, inquiries, and external references) are used along with measures of complexity to derive a data dimension count. The *functional dimension* is measured by considering "the number of internal operations required to transform input to output data".For the purposes of 3D function point computation, a "transformation" is viewed as a series of processing steps that are constrained by a set of semantic statements. The *control dimension* is measured by counting the number of transitions between states. A state represents some externally observable mode of behavior, and a transition occurs as a result of some event that causes the software or system to change its mode of behavior (i.e., to change state). For example, a wireless phone contains software that supports auto dial functions. To enter the *auto-dial* state from a *resting state,* the user presses an **Auto** key on the keypad. This event causes an LCD display to prompt for a code that will indicate the party to be called. Upon entry of the code and hitting the **Dial** key (another event), the wireless phone software makes a transition to the *dialing* state. When computing 3D function points, transitions are not assigned a complexity value.

To compute 3D function points, the following relationship is used:

index = $I + O + Q + F + E + T + R$

where *I, O, Q, F, E, T*, and *R* represent complexity weighted values for the elements discussed already: inputs, outputs, inquiries, internal data structures, external files, transformation, and transitions, respectively. Each complexity weighted value is computed using the following relationship:

complexity weighted value = $N_{il}W_{il} + N_{ia}W_{ia} + N_{ih}W_{ih}$ (4-3)

where N_{il}, N_{ia}, and N_{ih} represent the number of occurrences of element *i* (e.g., outputs) for each level of complexity (low, medium, high); and W_{il}, W_{ia}, and W_{ih} are the corresponding weights. The overall complexity of a transformation for 3D function points is shown in Figure 4.6.

It should be noted that function points, feature points, and 3D function points represent the same thing—"functionality" or "utility" delivered by software. In fact, each of these measures results in the same value if only the data dimension of an application is considered. For more complex real-time systems, the feature point count is often between 20 and 35 percent higher than the count determined using function points alone.

The function point (and its extensions), like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages; that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data; that counts of the information domain (and other dimensions) can be difficult to collect after the fact; and that FP has no direct physical meaning—it's just a number.

FIGURE 4.6
Determining the complexity of a transformation for 3D function points [WHI95].

Processing steps \ Semantic statements	1–5	6–10	11+
1–10	Low	Low	Average
11–20	Low	Average	High
21+	Average	High	High

METRICS FOR SOFTWARE QUALITY

The overriding goal of software engineering is to produce a high-quality system, application, or product. To achieve this goal, software engineers must apply effective methods coupled with modern tools within the context of a mature software process. In addition, a good software engineer (and good software engineering managers) must measure if high quality is to be realized.

The quality of a system, application, or product is only as good as the requirements that describe the problem, the design that models the solution, the code that leads to an executable program, and the tests that exercise the software to uncover errors. A good software engineer uses measurement to assess the quality of the analysis and design models, the source code, and the test cases that have been created as the software is engineered. To accomplish this real-time quality assessment, the engineer must use technical measures to evaluate quality in objective, rather than subjective ways.

The project manager must also evaluate quality as the project progresses. Private metrics collected by individual software engineers are assimilated to provide project level results. Although many quality measures can be collected, the primary thrust at the project level is to measure errors and defects. Metrics derived from these measures provide an indication of the effectiveness of individual and group software quality assurance and control activities.

Metrics such as work product (e.g., requirements or design) errors per function point, errors uncovered per review hour, and errors uncovered per testing hour provide insight into the efficacy of each of the activities implied by the metric. Error data can also be used to compute the defect removal efficiency (DRE) for each process framework activity.

- An Overview of Factors That Affect Quality

Over 25 years ago, McCall and Cavano [MCC78] defined a set of quality factors that were a first step toward the development of metrics for software quality. These factors assess software

from three distinct points of view: (1) product operation (using it), (2) product revision (changing it), and (3) product transition (modifying it to work in a different environment; i.e., "porting" it). In their work, the authors describe the relationship between these quality factors (what they call a framework) and other aspects of the software engineering process:

First, the framework provides a mechanism for the project manager to identify what qualities are important. These qualities are attributes of the software in addition to its functional correctness and performance which have life cycle implications. Such factors as maintainability and portability have been shown in recent years to have significant life cycle cost impact . . .

Secondly, the framework provides a means for quantitatively assessing how well the development is progressing relative to the quality goals established . . . Thirdly, the framework provides for more interaction of QA personnel throughout the development effort . . .

Lastly, . . . quality assurance personal can use indications of poor quality to help identify [better] standards to be enforced in the future.

- Measuring Quality

Although there are many measures of software quality, correctness, maintainability, integrity, and usability provide useful indicators for the project team. Gilb [GIL88] suggests definitions and measures for each.

1. Correctness. A program must operate correctly or it provides little value to its users. Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements. When considering the overall quality of a software product, defects are those problems reported by a user of the program after the program has been released for general use. For quality assessment purposes, defects are counted over a standard period of time, typically one year.
2. Maintainability. Software maintenance accounts for more effort than any other software engineering activity. Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. There is no way to measure maintainability directly; therefore, we must use indirect measures. A simple time-oriented metric is mean-time-to change (MTTC), the time it takes to analyze the change request, design a appropriate modification, implement the change, test it, and distribute the change to all users. On average, programs that are maintainable will have a lower MTTC (for equivalent types of changes) than programs that are not maintainable. Hitachi [TAJ81] has used a cost-oriented metric for maintainability called spoilage—the cost to correct defects encountered after the software has been released to its end-users. When the ratio of spoilage to overall project cost (for many projects) is plotted as a function of time, a manager can determine whether the overall maintainability of software produced by a software development organization is improving. Actions can then be taken in response to the insight gained from this information.
3. Integrity. Software integrity has become increasingly important in the age of hackers and firewalls. This attribute measures a system's ability to withstand attacks (both accidental and intentional) to its security. Attacks can be made on all three components of software:

programs, data, and documents. To measure integrity, two additional attributes must be defined: threat and security. Threat is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. Security is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as integrity = summation [(1 – threat) * (1 – security)] where threat and security are summed over each type of attack.

4. Usability. The catch phrase "user-friendliness" has become ubiquitous in discussions of software products. If a program is not user-friendly, it is often doomed to failure, even if the functions that it performs are valuable. Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics: (1) the physical and or intellectual skill required to learn the system, (2) the time required to become moderately efficient in the use of the system, (3) the net increase in productivity (over the approach that the system replaces) measured when the system is used by someone who is moderately efficient, and (4) a subjective assessment (sometimes obtained through a questionnaire) of users attitudes toward the system.

- Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner:

DRE = E/(E + D) (4-4)

where E is the number of errors found before delivery of the software to the end-user and D is the number of defects found after delivery.

The ideal value for DRE is 1. That is, no defects are found in the software. Realistically, D will be greater than 0, but the value of DRE can still approach 1. As E increases (for a given value of D), the overall value of DRE begins to approach 1. In fact, as E increases, it is likely that the final value of D will decrease (errors are filtered out before they become defects). If used as a metric that provides an indicator of the filtering ability of quality control and assurance activities, DRE encourages a software project team to institute techniques for finding as many errors as possible before delivery.

DRE can also be used within the project to assess a team's ability to find errors before they are passed to the next framework activity or software engineering task. For example, the requirements analysis task produces an analysis model that can be reviewed to find and correct errors. Those errors that are not found during the review of the analysis model are passed on to the design task (where they may or may not be found). When used in this context, we redefine DRE as

DREi = Ei/(Ei + Ei+1) (4-5)

where Ei is the number of errors found during software engineering activity i and Ei+1 is the number of errors found during software engineering activity i+1 that are traceable to errors that

were not discovered in software engineering activity i. A quality objective for a software team (or an individual software engineer) is to achieve DREi that approaches 1. That is, errors should be filtered out before they are passed on to the next activity.

METRICS FOR SMALL ORGANIZATIONS

The vast majority of software development organizations have fewer than 20 software people. It is unreasonable, and in most cases unrealistic, to expect that such organizations will develop comprehensive software metrics programs. However, it is reasonable to suggest that software organizations of all sizes measure and then use the resultant metrics to help improve their local software process and the quality and timeliness of the products they produce. Kautz [KAU99] describes a typical scenario that occurs when metrics programs are suggested for small software organizations:

Originally, the software developers greeted our activities with a great deal of skepticism, but they eventually accepted them because we kept our measurements simple, tailored them to each organization, and ensured that they produced valuable information. In the end, the programs provided a foundation for taking care of customers and for planning and carrying out future work.

What Kautz suggests is a commonsense approach to the implementation of any software process related activity: keep it simple, customize to meet local needs, and be sure it adds value. In the paragraphs that follow, we examine how these guidelines relate to metrics for small shops. “Keep it simple” is a guideline that works reasonably well in many activities. But how do we derive a “simple” set of software metrics that still provides value, and how can we be sure that these simple metrics will meet the needs of a particular software organization? We begin by focusing not on measurement but rather on results. The software group is polled to define a single objective that requires improvement. For example, “reduce the time to evaluate and implement change requests.” A small organization might select the following set of easily collected measures:

• Time (hours or days) elapsed from the time a request is made until evaluation is complete, tqueue.

• Effort (person-hours) to perform the evaluation, Weval.

• Time (hours or days) elapsed from completion of evaluation to assignment of

change order to personnel, teval.

• Effort (person-hours) required to make the change, Wchange.

• Time required (hours or days) to make the change, tchange.

• Errors uncovered during work to make change, Echange.

• Defects uncovered after change is released to the customer base, Dchange.

Once these measures have been collected for a number of change requests, it is possible to compute the total elapsed time from change request to implementation of the change and the percentage of elapsed time absorbed by initial queuing, evaluation

and change assignment, and change implementation. Similarly, the percentage of effort required for evaluation and implementation can be determined. These metrics can be assessed in the context of quality data, Echange and Dchange. The percentages provide insight into where the change request process slows down and may lead to process improvement steps to reduce tqueue, Weval, teval, Wchange, and/or Echange. In addition, the defect removal efficiency can be computed as

DRE = Echange / (Echange + Dchange)

DRE can be compared to elapsed time and total effort to determine the impact of quality assurance activities on the time and effort required to make a change. For small groups, the cost of collecting measures and computing metrics ranges from 3 to 8 percent of project budget during the learning phase and then drops to less than 1 percent of project budget after software engineers and project managers have become familiar with the metrics program.These costs can show a substantial return on investment if the insights derived from metrics data lead to meaningful process improvement for the software organization.

SOFTWARE PROJECT PLANNING

Software project planning involves estimation—your attempt to determine how much money, how much effort, how many resources, and how much time it will take to build a specific software-based system or product.

Software project management begins with a set of activities that are collectively called project planning. Before the project can begin, the manager and the software team must estimate the work to be done, the resources that will be required, and the time that will elapse from start to finish.

PROJECT PLANNING OBJECTIVES

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses. In addition, estimates should attempt to define best case and worst case scenarios so that project outcomes can be bounded.

The planning objective is achieved through a process of information discovery that leads to reasonable estimates

SOFTWARE SCOPE

The first activity in software project planning is the determination of software scope. Function and performance allocated to software during system engineering should be assessed to establish a project scope that is unambiguous and understandable at the management and technical levels. A statement of software scope must be bounded.

Software scope describes the data and control to be processed, function, performance, constraints, interfaces, and reliability. Functions described in the statement of scope are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of

decomposition is often useful. Performance considerations encompass processing and response time requirements. Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.

- Obtaining Information Necessary for Scope

Things are always somewhat hazy at the beginning of a software project. A need has been defined and basic goals and objectives have been enunciated, but the information necessary to define scope (a prerequisite for estimation) has not yet been delineated. The most commonly used technique to bridge the communication gap between the customer and developer and to get the communication process started is to conduct a preliminary meeting or interview. The first meeting between the software engineer (the analyst) and the customer can be likened to the awkwardness of a first date between two adolescents. Neither person knows what to say or ask, both are worried that what they do say will be misinterpreted; both are thinking about where it might lead (both likely have radically different expectations here); both want to get the thing over with; but at the same time, both want it to be a success.

Yet, communication must be initiated. Gause and Weinberg [GAU89] suggest that the analyst start by asking context-free questions; that is, a set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution desired, and the effectiveness of the first encounter itself.

The first set of context-free questions focuses on the customer, the overall goals and benefits. For example, the analyst might ask:

• Who is behind the request for this work?

• Who will use the solution?

• What will be the economic benefit of a successful solution?

• Is there another source for the solution?

The next set of questions enables the analyst to gain a better understanding of the problem and the customer to voice any perceptions about a solution:

• How would you (the customer) characterize "good" output that would be generated by a successful solution?

• What problem(s) will this solution address?

• Can you show me (or describe) the environment in which the solution will be used?

• Will any special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the meeting. Gause and Weinberg call these "meta-questions" and propose the following (abbreviated) list:

• Are you the right person to answer these questions? Are answers "official"?

• Are my questions relevant to the problem that you have?

• Am I asking too many questions?

• Can anyone else provide additional information?

• Should I be asking you anything else?

These questions (and others) will help to "break the ice" and initiate the communication that is essential to establish the scope of the project. But a question and answer meeting format is not an approach that has been overwhelmingly successful. In fact, the Q&A session should be used for the first encounter only and then be replaced by a meeting format that combines elements of problem solving, negotiation, and specification.

Customers and software engineers often have an unconscious "us and them" mindset. Rather than working as a team to identify and refine requirements, each constituency defines its own "territory" and communicates through a series of memos, formal position papers, documents, and question and answer sessions. History has shown that this approach works poorly. Misunderstandings abound, important information is omitted, and a successful working relationship is never established. With these problems in mind, a number of independent investigators have developed a team-oriented approach to requirements gathering that can be applied to help establish the scope of a project. Called facilitated application specification techniques (FAST), this approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of requirements.

Feasibility

Once scope has been identified (with the concurrence of the customer), it is reasonable to ask: “Can we build software to meet this scope? Is the project feasible?” All too often, software engineers rush past these questions (or are pushed past them by impatient managers or customers), only to become mired in a project that is doomed from the onset. Putnam and Myers address this issue when they write: . . . not everything imaginable is feasible, not even in software, evanescent as it may appear to outsiders. On the contrary, software feasibility has four solid dimensions: Technology— Is a project technically feasible? Is it within the state of the art? Can defects be reduced to a level matching the application’s needs? Finance—Is it financially feasible? Can development be completed at a cost the software organization, its client, or the market can afford?

Time—Will the project’s time-to-market beat the competition? Resources—Does the organization have the resources needed to succeed? For some projects in established areas the answers are easy. You have done projects

like this one before. After a few hours or sometimes a few weeks of investigation, you are sure you can do it again.

Projects on the margins of your experience are not so easy. A team may have to spend several months discovering what the central, difficult-to-implement requirements of a new application actually are. Do some of these requirements pose risks that would make the project infeasible? Can these risks be overcome? The feasibility team ought to carry initial architecture and design of the high-risk requirements to the point at which it can answer these questions. In some cases, when the team gets negative answers, a reduction in requirements may be negotiated.

Meantime, the cartoon people [senior managers] are drumming their fingers nervously on their large desks. Often, they wave their fat cigars in a lordly manner and yell impatiently

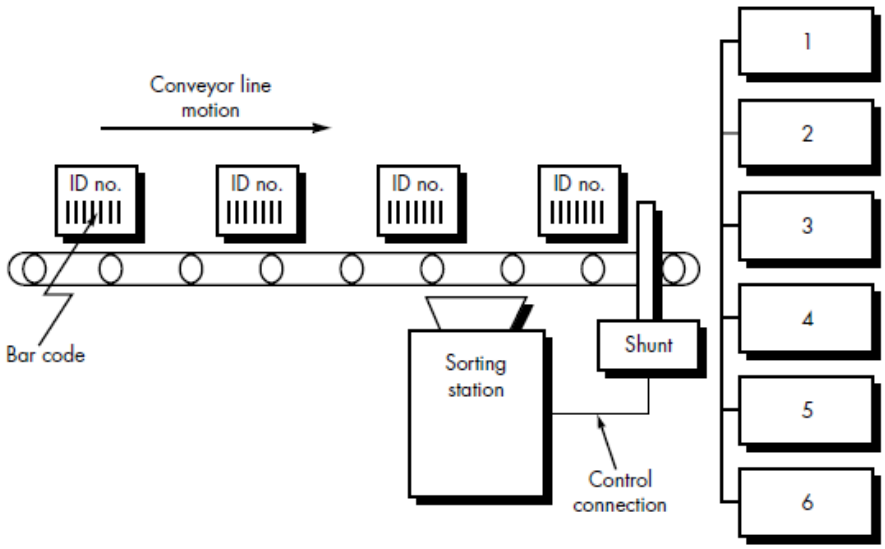
through the smoke screen, “Enough. Do it!” Many of the projects that appear in the newspapers a few years later as whopping failures got started this way.

- A Scoping Example

Communication with the customer leads to a definition of the data and control that are processed, the functions that must be implemented, the performance and constraints that bound the system, and related information. As an example, consider software for a conveyor line sorting system (CLSS). The statement of scope for CLSS follows:

The conveyor line sorting system (CLSS) sorts boxes moving along a conveyor line. Each box is identified by a bar code that contains a part number and is sorted into one of six bins at the end of the line. The boxes pass by a sorting station that contains a bar code reader and a PC. The sorting station PC is connected to a shunting mechanism that sorts the boxes into the bins. Boxes pass in random order and are evenly spaced. The line is moving at five feet per minute. CLSS is depicted schematically in Figure 5.1. CLSS software receives input information from a bar code reader at time intervals that conform to the conveyor line speed. Bar code data will be decoded into box identification format. The software will do a look-up in a part number database containing a maximum of 1000 entries to determine proper bin location for the box currently at the reader (sorting station). The proper bin location is passed to a sorting shunt that will position boxes in the appropriate bin. A record of the bin destination for each box will be maintained for later recovery and reporting. CLSS software will also receive input from a pulse tachometer that will be used to synchronize the control signal to the shunting mechanism. Based on the number of pulses generated between the sorting station and the shunt, the software will produce a control signal to the shunt to properly position the box.

FIGURE 5.1
A conveyor line sorting system



The project planner examines the statement of scope and extracts all important software functions. This process, called decomposition, was discussed in Chapter 3 and results in the following functions:

• Read bar code input.

• Read pulse tachometer.

• Decode part code data.

• Do database look-up.

• Determine bin location.

• Produce control signal for shunt.

• Maintain record of box destinations.

In this case, performance is dictated by conveyor line speed. Processing for each box must be completed before the next box arrives at the bar code reader. The CLSS software is constrained by the hardware it must access (the bar code reader, the shunt, the PC), the available memory, and the overall conveyor line configuration (evenly spaced boxes).

Function, performance, and constraints must be evaluated together. The same function can precipitate an order of magnitude difference in development effort when considered in the context of different performance bounds. The effort and cost required to develop CLSS software would be dramatically different if function remains the same (i.e., put boxes into bins) but performance varies. For instance, if the conveyor line average speed increases by a factor of 10 (performance) and boxes are no long spaced evenly (a constraint), software would become considerably more complex—thereby requiring more effort. Function, performance, and constraints are intimately connected. Software interacts with other elements of a computer-based system. The planner considers the nature and complexity of each interface to determine any effect on development resources, cost, and schedule. The concept of an interface is interpreted to include (1) the hardware (e.g., processor, peripherals) that executes the software and devices (e.g., machines, displays) indirectly controlled by the software, (2) software that already exists (e.g., database access routines, reusable software components, operating system) and must be linked to the new software, (3) people that make use of the software via keyboard or other I/O devices, and (4) procedures that precede or succeed the software as a sequential series of operations. In each case, the information transfer across the interface must be clearly understood.

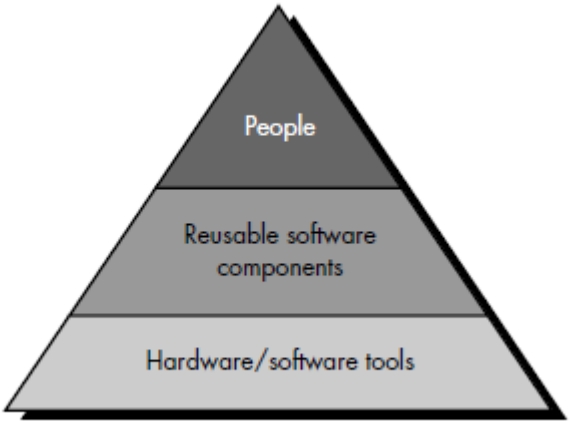
The least precise aspect of software scope is a discussion of reliability. Software reliability measures do exist but they are rarely used at this stage of a project. Classic hardware reliability characteristics like mean-time-between-failures (MTBF) can be difficult to translate to the software domain. However, the general nature of the software may dictate special considerations to ensure "reliability." For example, software for an air traffic control system or the space shuttle (both humanrated systems) must not fail or human life may be lost. An inventory control system or word-processor software should not fail, but the impact of failure is considerably less dramatic. Although it may not be possible to quantify software reliability as precisely as we would like in the statement of scope, we can use the nature of the project to aid in formulating estimates of effort and cost to assure reliability.

If a System Specification has been properly developed, nearly all information required for a description of software scope is available and documented before software project planning begins. In cases where a specification has not been developed, the planner must take on the role of system analyst to determine attributes and bounds that will influence estimation tasks.

RESOURCES

The second software planning task is estimation of the resources required to accomplish the software development effort. Figure 5.2 illustrates development resources as a pyramid. The development environment—hardware and software tools—sits at the foundation of the resources pyramid and provides the infrastructure to support the development effort. At a higher level, we encounter reusable software components— software building blocks that can dramatically reduce development costs and accelerate delivery. At the top of the pyramid is the primary resource—people. Each resource is specified with four characteristics: description of the resource, a statement of availability, time when the resource will be required; duration of time that resource will be applied. The last two characteristics can be viewed as a time window. Availability of the resource for a specified window must be established at the earliest practical time.

FIGURE 5.2
Project resources



Human Resources

The planner begins by evaluating scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client/server) are specified. For relatively small projects (one person-year or less), a single individual may perform all software engineering tasks, consulting with specialists as required. The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made

Reusable Software Resources

Component-based software engineering (CBSE)5 emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called components, must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

Bennatan suggests four software resource categories that should be considered as planning proceeds:

Off-the-shelf components. Existing software that can be acquired from a third party or that has been developed internally for a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.

Full-experience components. Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full-experience components will be relatively low-risk.

Partial-experience components. Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications required for partial-experience components have a fair degree of risk.

New components. Software components that must be built by the software team specifically for the needs of the current project.

The following guidelines should be considered by the software planner when reusable components are specified as a resource:

1. If off-the-shelf components meet project requirements, acquire them. The cost for acquisition and integration of off-the-shelf components will almost always be less than the cost to develop equivalent software.In addition, risk is relatively low.

2. If full-experience components are available, the risks associated with modification and integration are generally acceptable. The project plan should reflect the use of these components.

3. If partial-experience components are available, their use for the current project must be analyzed. If extensive modification is required before the components can be properly integrated with other elements of the software, proceed carefully—risk is high. The cost to modify partial-experience components can sometimes be greater than the cost to develop new components.

Ironically, reusable software components are often neglected during planning, only to become a paramount concern during the development phase of the software process. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur.

Environmental Resources

The environment that supports the software project, often called the software engineering environment (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice. Because most software organizations have multiple constituencies that require access to the SEE, a project planner must prescribe the time window required for hardware and software and verify that these resources will be available.

When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements being developed by other engineering teams. For example, software for a numerical control (NC) used on a class of machine tools may require a specific machine tool (e.g., an NC lathe) as part of the validation test step; a software project for advanced pagelayout may need a digital-typesetting system at some point during development. Each hardware element must be specified by the software project planner.

SOFTWARE PROJECT ESTIMATION

In the early days of computing, software costs constituted a small percentage of the overall computer-based system cost. An order of magnitude error in estimates of software cost had relatively little impact. Today, software is the most expensive element of virtually all computer-based systems. For complex, custom systems, a large cost estimation error can make the difference between profit and loss. Cost overrun can be disastrous for the developer.

Software cost and effort estimation will never be an exact science. Too many variables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it. However, software project estimation can be transformed from a black art to a series of systematic steps that provide estimates with acceptable risk.

To achieve reliable cost and effort estimates, a number of options arise:

1. Delay estimation until late in the project (obviously, we can achieve 100% accurate estimates after the project is complete!).

2. Base estimates on similar projects that have already been completed.

3. Use relatively simple decomposition techniques to generate project cost and effort estimates.

4. Use one or more empirical models for software cost and effort estimation.

Unfortunately, the first option, however attractive, is not practical. Cost estimates must be provided "up front." However, we should recognize that the longer we wait, the more we know, and the more we know, the less likely we are to make serious errors in our estimates.

The second option can work reasonably well, if the current project is quite similar to past efforts and other project influences (e.g., the customer, business conditions, the SEE, deadlines) are equivalent. Unfortunately, past experience has not always been a good indicator of future results.

The remaining options are viable approaches to software project estimation. Ideally, the techniques noted for each option should be applied in tandem; each used as a cross-check for the other. Decomposition techniques take a "divide and conquer" approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion. Empirical estimation models can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right. A model is based on experience (historical data) and takes the form

d = f (vi)

where d is one of a number of estimated values (e.g., effort, cost, project duration) and vi are selected independent parameters (e.g., estimated LOC or FP).

Automated estimation tools implement one or more decomposition techniques or empirical models. When combined with a graphical user interface, automated tools provide an attractive option for estimating. In such systems, the characteristics of the development organization (e.g., experience, environment) and the software to be developed are described. Cost and effort estimates are derived from these data.

Each of the viable software cost estimation options is only as good as the historical data used to seed the estimate. If no historical data exist, costing rests on a very shaky foundation.

DECOMPOSITION TECHNIQUES

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, we decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems.

the decomposition approach was discussed from two different points of view: decomposition of the problem and decomposition of the process. Estimation uses one or both forms of partitioning. But before an estimate can be made, the project planner must understand the scope of the software to be built and generate an estimate of its "size."

Software Sizing

The accuracy of a software project estimate is predicated on a number of things: (1) the degree to which the planner has properly estimated the size of the product to be,built; (2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects); (3) the degree to which the project plan reflects the abilities of the software team; and (4) the stability of product requirements and the environment that supports the software engineering effort. we consider the software sizing problem. Because a project estimate is only as good as the estimate of the size of the work to be accomplished, sizing represents the project planner's first major challenge. In the context of project planning, size refers to a quantifiable outcome of the software project. If a direct approach is taken, size can be measured in LOC. If an indirect approach is chosen, size is represented as FP.

Putnam and Myers suggest four different approaches to the sizing problem: "Fuzzy logic" sizing. This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.

Although personal experience can be used, the planner should also have access to a historical database of projects8 so that estimates can be compared to actual experience.

Function point sizing. The planner develops estimates of the information domain characteristics discussed already.

Standard component sizing. Software is composed of a number of different “standard components” that are generic to a particular application area.

For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to determine the delivered size per standard component. To illustrate, consider an information systems application. The planner estimates that 18 reports will be generated. Historical data indicates that 967 lines of COBOL are required per report. This enables the planner to estimate that 17,000 LOC will be required for the reports component. Similar estimates and computation are made for other standard components, and a combined size value (adjusted statistically) results.

Change sizing. This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished. Using an “effort ratio” for each type of change, the size of the change may be estimated.

Problem-Based Estimation

lines of code and function points were described as measures from which productivity metrics can be computed. LOC and FP data are used in two ways during software project estimation: (1) as an estimation variable to "size" each element of the software and (2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common. The project planner begins with a bounded statement of software scope and from this statement attempts to decompose software into problem functions that can each be estimated individually. LOC or FP (the estimation variable) is then estimated for each function. Alternatively, the planner may choose another component for sizing such as classes or objects, changes, or business processes affected.

Baseline productivity metrics (e.g., LOC/pm or FP/pm9) are then applied to the appropriate estimation variable, and cost or effort for the function is derived. Function estimates are combined to produce an overall estimate for the entire project. It is important to note, however, that there is often substantial scatter in productivity metrics for an organization, making the use of a single baseline productivity metric suspect. In general, LOC/pm or FP/pm averages should be computed by project domain. That is, projects should be grouped by team size, application area, complexity, and other relevant parameters. Local domain averages should then be computed. When a new project is estimated, it should first be allocated to a domain, and then the appropriate domain average for productivity should be used in generating the estimate.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The following decomposition approach has been adapted from Phillips

define product scope;

identify functions by decomposing scope;

do while functions remain

select a function j

assign all functions to subfunctions list;

do while subfunctions remain

select subfunctionk

if subfunctionk resembles subfunctiond described in a historical data base

then note historical cost, effort, size (LOC or FP) data for subfunctiond;

adjust historical cost, effort, size data based on any differences;

use adjusted cost, effort, size data to derive partial estimate, Ep;

project estimate = sum of {Ep};

else if cost, effort, size (LOC or FP) for subfunctionk can be estimated

then derive partial estimate, Ep;

project estimate = sum of {Ep};

else subdivide subfunctionk into smaller subfunctions;

add these to subfunctions list;

endif

endif

enddo

enddo

This decomposition approach assumes that all functions can be decomposed into subfunctions that will resemble entries in a historical data base. If this is not the case, then another sizing approach must be applied. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed.

For FP estimates, decomposition works differently. Rather than focusing on function, each of the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces—as well as the 14 complexity adjustment values discussed in Chapter 4 are estimated. The resultant estimates can then be used to derive a FP value that can be tied to past data and used to generate an estimate.

Regardless of the estimation variable that is used, the project planner begins by estimating range of values for each function or information domain value. Using historical data or (when all else fails) intuition, the planner estimates an optimistic, most likely, and pessimistic size value for each function or count for each information

domain value. An implicit indication of the degree of uncertainty is provided when a range of values is specified. A three-point or expected value can then be computed. The expected value for the estimation variable (size), S, can be computed as a weighted average of the optimistic (sopt), most likely (sm), and pessimistic (spess) estimates. For example,

$$S = (sopt + 4sm + spess)/6 \qquad (5\text{-}1)$$

gives heaviest credence to the “most likely” estimate and follows a beta probability distribution. We assume that there is a very small probability the actual size result will fall outside the optimistic or pessimistic values.

Once the expected value for the estimation variable has been determined, historical LOC or FP productivity data are applied. Are the estimates correct? The only reasonable answer to this question is: "We can't be sure." Any estimation technique, no matter how sophisticated, must be cross-checked with another approach. Even then, common sense and experience must prevail.

An Example of LOC-Based Estimation

As an example of LOC and FP problem-based estimation techniques, let us consider a software package to be developed for a computer-aided design application for mechanical components. A review of the System Specification indicates that the software is to execute on an engineering workstation and must interface with various computer graphics peripherals including a mouse, digitizer, high resolution color display and laser printer.

Using the System Specification as a guide, a preliminary statement of software scope can be developed: The CAD software will accept two- and three-dimensional geometric data from an engineer. The engineer will interact and control the CAD system through a user interface that will exhibit characteristics of good human/machine interface design. All geometric data and other supporting information will be maintained in a CAD database. Design analysis modules will be developed to produce the required output, which will be displayed on a variety of graphics devices. The software will be designed to control and interact with peripheral devices that include a mouse, digitizer, laser printer, and plotter.

This statement of scope is preliminary—it is not bounded. Every sentence would have to be expanded to provide concrete detail and quantitative bounding. For example, before estimation can begin the planner must determine what "characteristics of good human/machine interface design" means or what the size and sophistication of the "CAD database" are to be.

For our purposes, we assume that further refinement has occurred and that the following major software functions are identified:

• User interface and control facilities (UICF)

• Two-dimensional geometric analysis (2DGA)

• Three-dimensional geometric analysis (3DGA)

• Database management (DBM)

• Computer graphics display facilities (CGDF)

• Peripheral control function (PCF)

• Design analysis modules (DAM)

Following the decomposition technique for LOC, an estimation table, shown in Figure 5.3, is developed. A range of LOC estimates is developed for each function. For example, the range of LOC estimates for the 3D geometric analysis function is optimistic— 4600 LOC, most likely—6900 LOC, and pessimistic—8600 LOC.

Applying Equation (5-1), the expected value for the 3D geometric analysis function is 6800 LOC. Other estimates are derived in a similar fashion. By summing vertically in the estimated LOC column, an estimate of 33,200 lines of code is established for the CAD system. A review of historical data indicates that the organizational average productivity for systems of this type is 620 LOC/pm. Based on a burdened labor rate of $8000 per month, the cost per line of code is approximately $13. Based on the LOC estimate and the historical productivity data, the total estimated project cost is $431,000 and the estimated effort is 54 person-months

FIGURE 5.3 Estimation table for the LOC method

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

An Example of FP-Based Estimation

Decomposition for FP-based estimation focuses on information domain values rather than software functions. Referring to the function point calculation table presented in Figure 5.4, the project planner estimates inputs, outputs, inquiries, files, and external interfaces for the CAD software. For the purposes of this estimate, the complexity weighting factor is assumed to be average. Figure 5.4 presents the results of this estimate.

Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of inputs	20	24	30	24	4	97
Number of outputs	12	15	22	16	5	78
Number of inquiries	16	22	28	22	5	88
Number of files	4	4	5	4	10	42
Number of external interfaces	2	2	3	2	7	15
Count total						*320*

Each of the complexity weighting factors is estimated and the complexity adjustment factor is computed Finally, the estimated number of FP is derived:

FPestimated = count-total x [0.65 + 0.01 x (Fi)]

FPestimated = 375

The organizational average productivity for systems of this type is 6.5 FP/pm. Based on a burdened labor rate of $8000 per month, the cost per FP is approximately $1230. Based on the LOC estimate and the historical productivity data, the total estimated project cost is $461,000 and the estimated effort is 58 person-months.

Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
On-line data entry	4
Input transaction over multiple screens	5
Master files updated on-line	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5
Complexity adjustment factor	1.17

Process-Based Estimation

The most common technique for estimating a project is to base the estimate on the process that will be used. That is, the process is decomposed into a relatively small set of tasks and the effort required to accomplish each task is estimated. Like the problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope. A series of software process activities must be performed for each function. Functions and related software process activities may be represented as part of a table similar to the one presented in Figure 3.2.

Once problem functions and process activities are melded, the planner estimates the effort (e.g., person-months) that will be required to accomplish each software process activity for each software function. These data constitute the central matrix of the table in Figure 3.2. Average labor rates (i.e., cost/unit effort) are then applied to the effort estimated for each process activity. It is very likely the labor rate will vary for each task. Senior staff heavily involved in early activities are generally more expensive than junior staff involved in later design tasks, code generation, and early testing.

Costs and effort for each function and software process activity are computed as the last step. If process-based estimation is performed independently of LOC or FP estimation, we now have

two or three estimates for cost and effort that may be compared and reconciled. If both sets of estimates show reasonable agreement, there is good reason to believe that the estimates are reliable. If, on the other hand, the results of these decomposition techniques show little agreement, further investigation and analysis must be conducted.

An Example of Process-Based Estimation

To illustrate the use of process-based estimation, we again consider the CAD software introduced in Section 5.6.3. The system configuration and all software functions remain unchanged and are indicated by project scope.

Referring to the completed process-based table shown in Figure 5.5, estimates of effort (in person-months) for each software engineering activity are provided for each CAD software function (abbreviated for brevity). The engineering and construction release activities are subdivided into the major software engineering tasks shown. Gross estimates of effort are provided for customer communication, planning, and risk analysis. These are noted in the total row at the bottom of the table. Horizontal and vertical totals provide an indication of estimated effort required for analysis, design, code, and test. It should be noted that 53 percent of all effort is expended on front-end engineering tasks (requirements analysis and design), indicating the relative importance of this work.

Based on an average burdened labor rate of $8,000 per month, the total estimated project cost is $368,000 and the estimated effort is 46 person-months. If desired, labor rates could be associated with each software process activity or software engineering task and computed separately.

Total estimated effort for the CAD software range from a low of 46 person-months (derived using a process-based estimation approach) to a high of 58 person-months (derived using an FP estimation approach). The average estimate (using all three approaches) is 53 person-months. The maximum variation from the average estimate is approximately 13 percent.

What happens when agreement between estimates is poor? The answer to this question requires a re-evaluation of information used to make the estimates. Widely divergent estimates can often be traced to one of two causes:

1. The scope of the project is not adequately understood or has been misinterpreted by the planner.

2. Productivity data used for problem-based estimation techniques is inappropriate for the application, obsolete (in that it no longer accurately reflects the software engineering organization), or has been misapplied.

The planner must determine the cause of divergence and then reconcile the estimates.

AUTOMATED ESTIMATION TOOLS

The decomposition techniques and empirical estimation models described in the preceding sections are available as part of a wide variety of software tools. These automated estimation tools allow the planner to estimate cost and effort and to perform "what-if" analyses for important project variables such as delivery date or staffing.

Although many automated estimation tools exist, all exhibit the same general characteristics and all perform the following six generic functions [JON96]:

1. Sizing of project deliverables. The “size” of one or more software work products is estimated. Work products include the external representation of software (e.g., screen, reports), the software itself (e.g., KLOC), functionality delivered (e.g., function points), descriptive information (e.g. documents).

2. Selecting project activities. The appropriate process framework is selected and the software engineering task set is specified.

3. Predicting staffing levels. The number of people who will be available to do the work is specified. Because the relationship between people available and work (predicted effort) is highly nonlinear, this is an important input.

4. Predicting software effort. Estimation tools use one or more models that relate the size of the project deliverables to the effort required to produce them.

5. Predicting software cost. Given the results of step 4, costs can be computed by allocating labor rates to the project activities noted in step 2.

6. Predicting software schedules. When effort, staffing level, and project activities are known, a draft schedule can be produced by allocating labor across software engineering activities based on recommended models for effort distribution

When different estimation tools are applied to the same project data, a relatively large variation in estimated results is encountered. More important, predicted values sometimes are significantly different than actual values. This reinforces the notion that the output of estimation tools should be used as one "data point" from which estimates are derived—not as the only source for an estimate.

RISK ANALYSIS AND MANAGEMENT

Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty. Many problems can plague a software project. A risk is potential problem—it might happen, it might not. But, regardless of the outcome, it’s a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur

REACTIVE VS. PROACTIVE RISK STRATEGIES

Reactive risk strategies have been laughingly called the “Indiana Jones school of risk management” [THO92]. In the movies that carried his name, Indiana Jones, when faced with overwhelming difficulty, would invariably say, “Don’t worry, I’ll think of something!” Never worrying about problems until they happened, Indy would react in some heroic way. Sadly, the average software project manager is not Indiana Jones and the members of the software project team are not his trusty sidekicks. Yet, the majority of software teams rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does

nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a fire fighting mode. When this fails, “crisis management takes over and the project is in real jeopardy.

A considerably more intelligent strategy for risk management is to be proactive. A proactive strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner. Throughout the remainder of this chapter, we discuss a proactive strategy for risk management.

SOFTWARE RISKS

Although there has been considerable debate about the proper definition for software risk, there is general agreement that risk always involves two characteristics

• Uncertainty—the risk may or may not happen; that is, there are no 100% probable risks.1

• Loss—if the risk becomes a reality, unwanted consequences or losses will occur.

When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project risks threaten the project plan. That is, if project risks become real, it is likely that project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project.

project complexity, size, and the degree of structural uncertainty were also defined as project (and estimation) risk factors. Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than we thought it would be.

Business risks threaten the viability of the software to be built. Business risks often jeopardize the project or the product. Candidates for the top five business risks are (1) building a excellent product or system that no one really wants (market risk), (2) building a product that no longer fits into the overall business strategy for the company (strategic risk), (3) building a product that the sales force doesn't understand how to sell, (4) losing the support of senior management due to a change in focus or a change in people (management risk), and (5) losing budgetary or personnel commitment (budget risks). It is extremely important to note that simple categorization won't always work. Some risks are simply unpredictable in advance.

Another general categorization of risks has been proposed by Charette.

Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment). Predictable

risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced). Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

RISK IDENTIFICATION

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories that have been presented generic risks and product-specific risks. Generic risks are a potential threat to every software project. Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the project at hand. To identify product-specific risks, the project plan and the software statement of scope are examined and an answer to the following question is developed: "What special characteristics of this product may threaten our project plan?"

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

• Product size—risks associated with the overall size of the software to be built or modified.

• Business impact—risks associated with constraints imposed by management or the marketplace.

• Customer characteristics—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.

• Process definition—risks associated with the degree to which the software process has been defined and is followed by the development organization.

• Development environment—risks associated with the availability and quality of the tools to be used to build the product.

• Technology to be built—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.

• Staff size and experience—risks associated with the overall technical and project experience of the software engineers who will do the work.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answers to these questions allow the planner to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of “risk components and drivers" [AFC88] are listed along with their probability of occurrence. Drivers for performance, support, cost, and schedule are discussed in answer to later questions.

A number of comprehensive checklists for software project risk have been proposed in the literature. These provide useful insight into generic risks for software projects and should be used whenever risk analysis and management is instituted. However, a relatively short list of questions can be used to provide a preliminary indication of whether a project is "at risk."

- Assessing Overall Project Risk

The following questions have derived from risk data obtained by surveying experienced software project managers in different part of the world [KEI98]. The questions are ordered by their relative importance to the success of a project.

1. Have top software and customer managers formally committed to support the project?

2. Are end-users enthusiastically committed to the project and the system/product to be built?

3. Are requirements fully understood by the software engineering team and their customers?

4. Have customers been involved fully in the definition of requirements?

5. Do end-users have realistic expectations?

6. Is project scope stable?

7. Does the software engineering team have the right mix of skills?

8. Are project requirements stable?

9. Does the project team have experience with the technology to be implemented?

10. Is the number of people on the project team adequate to do the job?

11. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

If any one of these questions is answered negatively, mitigation, monitoring, and management steps should be instituted without fail. The degree to which the project is at risk is directly proportional to the number of negative responses to these questions.

Risk Components and Drivers

The U.S. Air Force [AFC88] has written a pamphlet that contains excellent guidelines for software risk identification and abatement. The Air Force approach requires that the project manager identify the risk drivers that affect software risk components— performance, cost, support, and schedule. In the context of this discussion, the risk components are defined in the following manner:

• Performance risk—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.

• Cost risk—the degree of uncertainty that the project budget will be maintained.

• Support risk—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.

• Schedule risk—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

The impact of each risk driver on the risk component is divided into one of four impact categories—negligible, marginal, critical, or catastrophic. Referring to Figure 6.1 a characterization of the potential consequences of errors (rows labeled 1) or a failure to achieve a desired outcome (rows labeled 2) are described. The impact category is chosen based on the characterization that best fits the description in the table.

Components / Category		Performance	Support	Cost	Schedule
Catastrophic	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of $500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable IOC
Critical	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of $100K to $500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in IOC
Marginal	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of $1K to $100K	
	2	Minimal to small reduction in technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
Negligible	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than $1K	
	2	No reduction in technical performance	Easily supportable software	Possible budget underrun	Early achievable IOC

Note: (1) The potential consequence of undetected software errors or faults.
(2) The potential consequence if the desired outcome is not achieved.

FIGURE 6.1 Impact assessment [BOE89]

RISK PROJECTION

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.2 A sample risk table is illustrated in Figure 6.2.

Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
•				
•				
•				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

FIGURE 6.2 Sample risk table prior to sorting

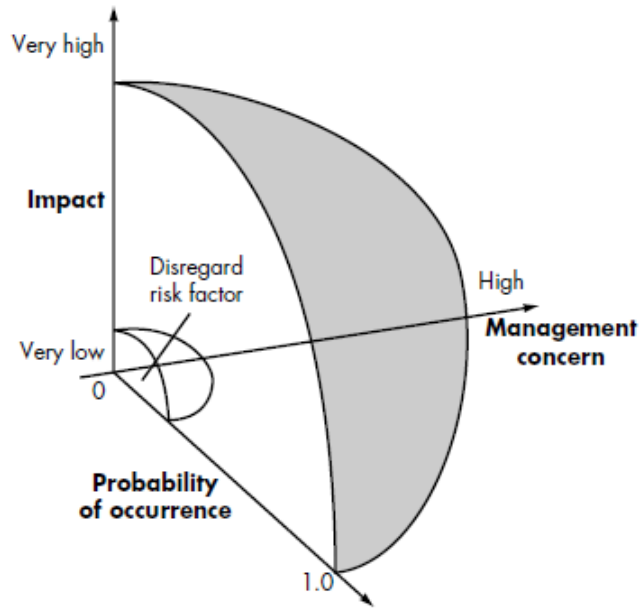


FIGURE 6.3 Risk and management concern

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 referenced in Section 6.3. 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 using the characterization presented in Figure 6.1, 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. Referring to Figure 6.3, 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 labelled 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. The RMMM plan and risk information sheets are discussed in Sections 6.5 and 6.6.

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 [AFC88]. 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 [AFC88], 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. Using Figure 6.1, determine the impact for each component based on the criteria shown.

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 described 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 [CHA89] 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. Figure 6.4 represents this situation graphically. 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.

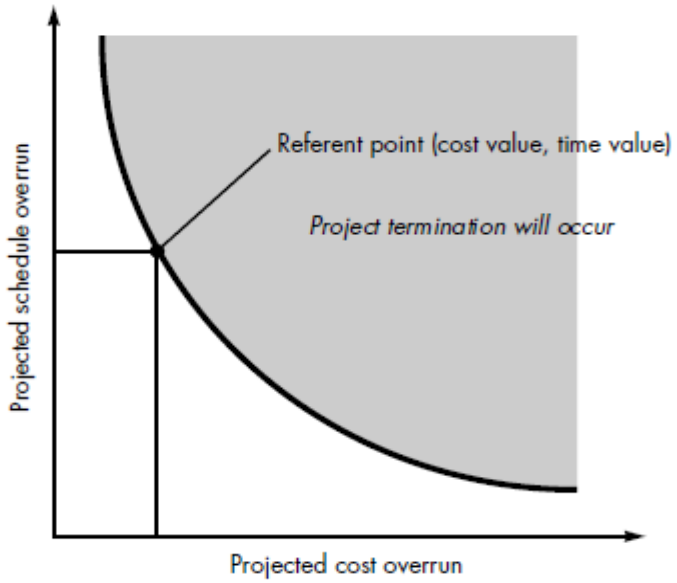
2. Attempt to develop a relationship between each (ri, li, xi) and each of the referent levels.

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.

A detailed discussion of risk referent level is best left to books that are dedicated to risk analysis

FIGURE 6.4
Risk referent level



RISK REFINEMENT

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage.

One way to do this is to represent the risk in condition-transition-consequence (CTC) format [GLU94]. That is, the risk is stated in the following form:

Given that <condition> then there is concern that (possibly) <consequence>.

Using the CTC format for the reuse risk noted in Section 6.4.2, we can write: Given that all reusable software components must conform to specific design standards and that some do not conform, then there is concern that (possibly) only 70 percent of the planned reusable modules may actually be integrated into the as-built system, resulting in

the need to custom engineer the remaining 30 percent of components. This general condition can be refined in the following manner:

Subcondition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.

Subcondition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

Subcondition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

The consequences associated with these refined subconditions remains the same (i.e., 30 percent of software components must be customer engineered), but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

THE RMMM PLAN

A risk management strategy can be included in the software project plan or the risk management steps can be organized into a separate Risk Mitigation, Monitoring and Management Plan. The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan. Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a risk information sheet (RIS) [WIL97]. In most cases, the RIS is maintained using a database system, so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily. The format of the RIS is illustrated in Figure 6.5.

Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence. As we have already discussed, risk mitigation is a problem avoidance activity. Risk monitoring is a project tracking activity with three primary objectives: (1) to assess whether predicted risks do, in fact, occur; (2) to ensure that risk aversion steps defined for the risk are being properly applied; and (3) to collect information that can be used for future risk analysis. In many cases, the problems that occur during a project can be traced to more than one risk. Another job of risk monitoring is to attempt to allocate origin (what risk(s) caused which problems throughout the project).