CHAPTER 7

Size, Effort, and Scheduling of Projects

Importance of Size

A person intending to build a house typically estimates the overall size of the house in number of square feet. While buying an office table, you may specify the size as *Length* × *Breadth* × *Height*. Almost every object used in daily life can be sized by using one or more parameters. Because software is "soft," it is always quite difficult to size it as accurately as other material products like a house, a car, or a television. Software professionals traditionally have been measuring the size of software applications by using different methods; Lines-Of-Code (LOC) is the oldest and most popular method used.

Whatever the level of approximation, LOC does give some sense of software size. For example, you can differentiate between two applications developed in the same language that have LOC figures of 20,000 and 50,000. Measurement of software size (in LOC or other units) is as important to a software professional as measurement of a building (in square feet) is to a building contractor. All other derived data, including effort to deliver a software project, delivery schedule, and cost of the project, are based on one of its major input elements: software size.

Key Inputs to Software Sizing

In software estimation parlance, scope of work (also expressed in terms of business functionality provided) is one of the key inputs that determine

the size of the final product being delivered. Software estimators sometimes confuse *size* and *effort*. **Size,** in a software development context, is the complete set of business functionalities that the end user gets when the product is deployed and in use. And the person months required to produce the software application of a given size is the **effort**. For example, in an invoicing application, the user gets the capability to prepare invoices for products sold to customers. The features provided in the invoicing application are equivalent to the functionality available; hence, the size of the application. The amount of time spent by the programmers to design and build the invoicing application is the effort. IT organizations should practice recording metrics of past sizing activities and use the information and experience to size future projects. In his technical paper, "Size Does Matter: Continuous Size Estimating and Tracking," Mike Ross observes, "The secret to making progress in sizing these new environments (software applications) is to identify the *unit of human thought* in the abstraction being used. Next, the organization must go through a calibration process, starting with projects for which the actual size can be determined in terms of that *unit of human thought*" [1].

Differentiate Functions from Production Effort/Costs

While negotiating the price of a product, both the buyer and the seller normally assume that the final price includes all aspects of the product; that is, features (functions), look and feel (user interface), quality and reliability (performance), and more. When two products of the same type (family) are being compared, very rarely are products 100 percent identical. To illustrate, compare any two products (for example, mobile phones like Nokia and Samsung, laptops like IBM and Toshiba, or software products like Oracle and Microsoft SQL Server); none of these products are the same in all respects. Similarly, each software project has its own unique requirements, and as such, estimating the cost of the project will also be unique if you adopt the process of assessing effort based on software features, such as quality, scalability, and reliability, which are normally unique for each project.

Estimating the effort and cost of software development projects is perhaps much more complex than estimating the production costs of most consumer products as well as other areas of project execution, whether it involves construction, manufacturing, services, or other elements. Table 7.1 provides some insight into the key differences between a typical product development and software development activities.

Table 7.1 *Activity Comparison for Products and Software Development*

#	*Activity Description*	*Typical Product/Service*	*Software Development*
1	Functions/ Features Clarity	Very clear, defined, and definite	Somewhat vague; can and will change during development phase
2	Quality attributes	Accurately measurable	Measurable but cost of quality varies based on team skills
3	Tools availability	Well-established tools available in the market	Tools with somewhat ambiguous claims on productivity gains
4	Skills/ Competency of workers	Defined	Defined
5	Production effort	Can be estimated very well	Quite loosely estimated
6	Effort/Cost Estimation	Definite	Gut feel + Buffer + Contingency; typically overruled by managers
7	Changes during production	Negligible	Very frequent
8	Specification & Design	Well-defined, reviewed, and signed-off before start of production	Loosely defined, keeps changing throughout the lifecycle of product development
9	Rework/ Improvements	Almost impossible to modify once the product is delivered	Always possible to modify even after it goes into production

Over and above the ambiguity and ever-changing scope (functionality and features) in the software development projects shown in Table 7.1, if you add issues related to the target technology platform on which the software is being developed, the cup of woes would be full to the brim. No wonder that the question that bothers a software development

project team is, "Can there be an alternative and better-defined method of estimating project execution effort and costs?" On second thought, wouldn't it be wonderful to have a standard yardstick that can measure different products with different sets of functions against the same measuring scale? This yardstick should provide the user with a true comparison facility. For example, a measuring tape can measure a table size, the number of square feet in a house, the height of a tower, the distance between two locations, and more. All the items are different in nature but the measuring unit (yardstick) is same. Further, if you measure two similar items (such as the distance between locations) using the same measuring tape, you can compare the two measurements and even define a relative size ratio between them.

Function Point Analysis Method

The Function Point Analysis (FPA) methodology-based estimation model designed by Allan Albrecht of IBM in 1979, and now owned and continuously upgraded by IFPUG [2] (International Function Point Users Group), is perhaps the nearest to separating the functions delivered by a product from the technology platform on which the product is developed and hence the path to deriving the total effort and cost to deliver the application. The uniqueness of this FPA method enables the estimator to clearly size the software application (product) based purely on the functions that are expected to be delivered by the application. Perhaps it is due to this unique feature in the FPA method that its popularity and usage, as compared to other estimation methods, is the highest in the software developer community.

To understand the uniqueness of the FPA method, consider the example of a mobile phone, as shown in Figure 7.1. From a mobile phone user's perspective, the various functions the user expects to experience are

- To be able to communicate with contacts, friends, and family at will, irrespective of physical location and environment
- Instant, online access to the directory of contact numbers
- Provision to send SMS (text) messages to anyone, any time
- Internet browsing facility
- Storing and playing music

The FPA method is built on the premise that the function point (FP) count finally determined is based totally on the user's perspective of the functions expected to be delivered with the final product.

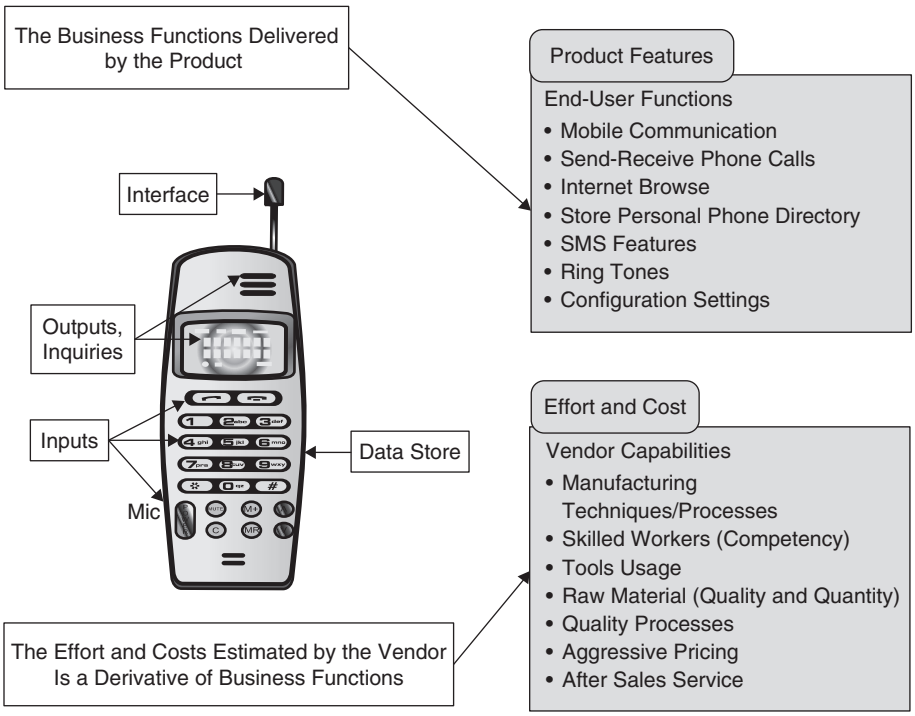

Figure 7.1 *Defining size.*

In Figure 7.1, the Product Features map clearly to the end user functions that are expected to be delivered from the product (mobile phone). The FP counting method provides key inputs and maps to this aspect of Product Size.

Now consider the other half of the effort and cost-calculation activities (other than Product Sizing), which contribute toward arriving at the overall product pricing. Figure 7.1 shows those activities as:

- Manufacturing techniques/processes
- Skills/Competency of the workers
- Effective usage of right tools
- Raw material
- Quality process
- Pricing
- After sales service

A careful review of these parameters exposes the fact that most of these activities and processes are vendor specific and depend on the team assembled to deliver the project. The Effort and Cost text box contains all activities that point to vendor capabilities. Understanding the two different attributes of a production activity in terms of Product Size and Effort and Cost paves the way for further discussion and focus on Size as the single, critical measurement attribute in software development projects.

Size—The Differentiator

In a typical software development project scenario, the end user (or business user) is the owner (customer) of the application being developed and the IT development team or an external (outsourced) vendor is the supplier. Knowing the size of the application would come in very handy for the customer in situations where an evaluation of multiple vendors, including the internal IT development team, is being done. Here are the key pointers to successful negotiation of software development contracts:

- When the application's size is predetermined, the user can avoid being misled by different vendors on various functional complexities of the application. Instead, the vendors could be asked to provide their quotes for the defined size of the application being developed on a given technology platform.
- The user may not have a deep knowledge of the internals of the application development process or even the technology involved (sometimes the technical architecture and coding of a project is compared to a black box). Despite this situation, the user can still manage all project contract execution activities based on the final size of the product that needs to be delivered and accepted.
- "An experienced estimator can produce reasonably good size estimates by analogy if accurate size values are available for the previous project and if the new project is sufficiently similar to the previous one" [3].
- "Basically, function points are the quantified representation—or size—of the functions (or transactions) and data that go together to operate as a computer application. They provide a rough unit of size that is based upon the *software component* of the requirement" [4].
- Because functional features are separated through the size model, this opens the opportunity for the customer to choose the technology platform on which the application needs to be developed.

Development costs on different technology vary based on skills available, and this leads to better control over cost and budget constraints.

- Function points are perhaps one of the best methods to estimate the size of an application. The method is quite ambiguous and therefore flexible enough to be molded into a variety of estimation needs, such as software development, maintenance, reengineering, enhancement, etc. Source Lines of Code (SLOC) or LOC is a poor alternative.
- "…'*Size*' refers in a very general way to the total scope of a program. It includes the breadth and depth of the feature set as well as the program's difficulty and complexity" [5].

The Yardstick

The business value delivered by software systems, in the form of functions provided, is one of the key inputs to the size of the system. This size can be used as an effective yardstick to facilitate many needs of an IT organization. In much the same way a measuring tape can be used to measure height, length, width, and depth of a variety of material objects and places, size can be an effective yardstick for software estimation in projects ranging from simple to complex.

Inputs to Sizing

The size of a software application can be measured in several measuring units. Popular sizing units include Source Lines of Code (SLOC), function points (FP), object points, and feature points. Each measuring unit is unique in the sense that the measuring unit defined by the SLOC method does not have a clear and measurable equivalent ratio with, for example, function points or object points. Each measuring unit also measures the application size from a different perspective. For example,

- SLOC measures the total lines of code that can be counted in the software application. The SLOC are specific to the technology platform on which the application was developed. For a given application, the SLOC count will vary for different technology platforms. The number of SLOC, if written in COBOL, would be far higher when compared to SLOC written in C, although the delivered functionality would be the same in both cases.

- Function points (FP) count measures the business functionality delivered by the application being counted. The size measured through FP method is independent of the technology on which the application is developed. Therein lies the comforting fact that when two different experienced estimators are given an application, the size they measure in function points will be within a comfortable variance range.
- Object points measures the size of an application by counting the number of screens, reports, and interfaces (known as *objects*) required to complete the coding. The objects themselves can be classified into different level of complexity such as simple, average, and complex.
- Feature points [6] is a variant of function points. Developed by Capers Jones, this method adds another measurable attribute, the complexity of algorithms. Here the size is measured in feature points count very similar to function point count.

Source of Inputs

For any type of estimation, it is important to have detailed information about the application/system. This is also the case with function point estimation. For estimating the size of an application using function point analysis, the following information about the application is required:

- Scope (specifications) of the application
- Data being maintained by the application
- Data being referenced (but not being maintained) by the application
- Logical grouping of the data being used by the application
- Input to the application, in terms of data within the defined application scope
- Output from the application like reports, queries, etc., within the scope of application requirements
- The general behavioral characteristics of the system

These inputs can be categorized into two distinct types of requirements, typically called *functional* and *non-functional* requirements. The requirements information can be obtained from the following artifacts:

- Functional requirements:
  - Business process
  - Business work flow

  - Conceptual data model
  - Functional specifications or use case specifications
  - Input on external applications, if any, including details of interface with other applications
- Non-functional requirements:
  - Documents stating the performance requirements and other infrastructure details, like multiple sites, reliability, scalability, availability, usability, etc.

Accuracy of Requirements

The quality and accuracy of the requirements specification is critical to correct sizing. The specifications need to be complete, correct, and up-to-date for the size calculation. The result of wrong size estimation will lead to incorrect effort projections, schedule calculations, and other project planning and project costs because these activities are based heavily on the size information.

Details of all the functionality required by the user at the time of project initiation may not be available, which can lead to scope change later (also known as *scope creep*). The specifications should include the business data and reference data. Following are some of the points that must be specified to ensure the accuracy of the requirements specifications documents:

- Objectives of the proposed system
- Business requirements
- Scope and boundary
- Business events/functions
- Screens and reports
- External interfaces
- Operating environment; hardware, operating system, technology platform, and network protocols
- Specific communication requirements
- Performance requirements
- Special user requirements
- Data migration
- Architectural requirements
- Constraints/issues

Role of Size in the Software Development Lifecycle

The purpose of evaluating and estimating the size of a software application is to move the estimation process forward to arrive at effort, schedule, and costs of executing the project. As such, it is essential for the estimator to understand the relation between the various categories of size units (SLOC, FP, object points, and feature points) to its applicability in the software development lifecycle stages. The typical lifecycle stages of software project execution considered here are (1) the requirement phase, (2) the design phase, (3) the build (construction) phase, and (4) the test phase. Consider the following points:

- The Source Lines-Of-Code (SLOC) is the most ambiguous of all the sizing units and is highly unreliable as far as the estimation process is considered. Obviously, estimations are done when the application is yet to be developed. Predicting the SLOC that the application will generate upon project completion would be quite an uphill task.
- The other concern is to relate the SLOC count to the lifecycle stages of software project execution. Although the total SLOC generated at the end of project completion is measurable, coming up with an estimation formula that would give you the break-up effort for various lifecycle stages of project execution individually would be complex. To better illustrate, consider an application in COBOL that generates 50,000 SLOC. How do you relate this SLOC count to the effort it would take to complete the lifecycles stages (requirements, design, build, and test)? Experts in COBOL may be able to provide an approximate effort that it takes to code and test a given count of SLOC. The estimator will then have to convert the code and test (build) effort to the total project execution effort through extrapolation methods. Every bit of error in calculation of code and test effort will be multiplied during the extrapolation.
- Function points is perhaps the nearest method to providing a measuring unit that spans across all the standard lifecycle phases of software project execution. Function point count of an application under development depicts the total functionality that the user expects to be delivered. If you have the data on the delivery rate of the project team, the total project effort can thus be calculated by dividing the FP count by the delivery rate (productivity). Further, the total project execution effort can be now divided into lifecycle phase efforts based either on historic data from your own organization or industry standard data.

For example, consider a project that has a count of 1,000 FP. Assuming a delivery rate (productivity) of the project team on a selected platform to be 15 FP per person month, you get a total effort of approximately 67 person months (1,000 FP divided by 15 FP per person month). If you assume a lifecycle phase breakup of effort as

- *Requirements:* 15 percent
- *Design:* 20 percent
- *Build:* 40 percent
- *Test:* 25 percent

The total project effort can be divided according to the preceding percentages. The percentage breakup of effort can be fine-tuned by taking feedback from completed projects and applying corrections as applicable through continuous improvement cycles.

- Feature points are quite similar to function points.
- Object points looks at the number of objects (screens, reports, etc.) likely to be generated in the software. The effort estimated for objects is similar to that of SLOC and it is focused on the code and test phase of the project execution lifecycle. The full effort is calculated by extrapolation as is done in the case of SLOC estimation.

Impact of Delivery Rate

Sizing a software project is important, but knowing the delivery capacity of your programmers is even more critical. The foundation of a predictable project execution is the ability to attain repeatable delivery processes. Having a good handle on the IT organization's delivery rate helps IT managers commit to project delivery timelines with higher confidence.

Measuring the actual productivity of the programmers in an organization is a lengthy and iterative process. Depending on the size of the IT organization and the variety of technology platforms in use, the definition of productivity becomes more and more complex. The following sections discuss some of these complexities and variations.

Productivity Drivers

Productivity, sometimes known as *delivery rate*, can be viewed as the ratio of output/input. In a software development parlance, output is

typically the software product and input is the effort spent by the programmers. The three major ingredients that define productivity are

- Software product (application)
- Software production process (the software project lifecycle activities)
- Software development environment

The productivity measuring team should be able to define and measure each of these ingredients in order to be able to calculate delivery rate figures.

Software Product

The product that is the final delivered output is also called the **software application**. Other secondary outputs that are delivered during a typical software development project are documentation for requirement specification, design, test plans, test results, and operations manual. In project management, these are called *project deliverables*.

Software Production Process

Various activities that take place during typical software development lifecycle stages need different process definition. Typical lifecycle activities are

- Requirement analysis and specification
- Architecture
- Detailed design
- Build and unit test
- System and integration test

Different activities in each lifecycle stage need varied skills and competencies. Even with a well-defined process for executing each of the lifecycle activities, the competency of the individual resource controls the productivity for the lifecycle activity.

Software Development Environment

The general environment under which the software project team works contributes toward significant variation in productivity. A good development environment supported by tested and productive tools plays a big role in productivity improvements of the project team. A robust quality control mechanism backed further by reusable software components, artifacts, and best practices further enhances productivity rates.

This is perhaps the most difficult part of setting up a well-structured IT organization. It takes considerable length of time to achieve the maturity in defining, deploying, and accepting a wide range of highly productive tools and an equally large number of reusable components that lead to high productivity figures.

Productivity Measurement

Among the three ingredients that impact software development productivity (the product, the development resources and processes, and the environment), the output is the software product and the input is the effort spent during software production stages. The environment, under which the production takes place, controls the variation in input efforts.

Figure 7.2 shows that different lifecycle stage activities require resources with an appropriate level of competency. The most common roles involved in a typical software development project are

- Project manager
- Technical architect
- Business analyst
- Programmers
- Testers

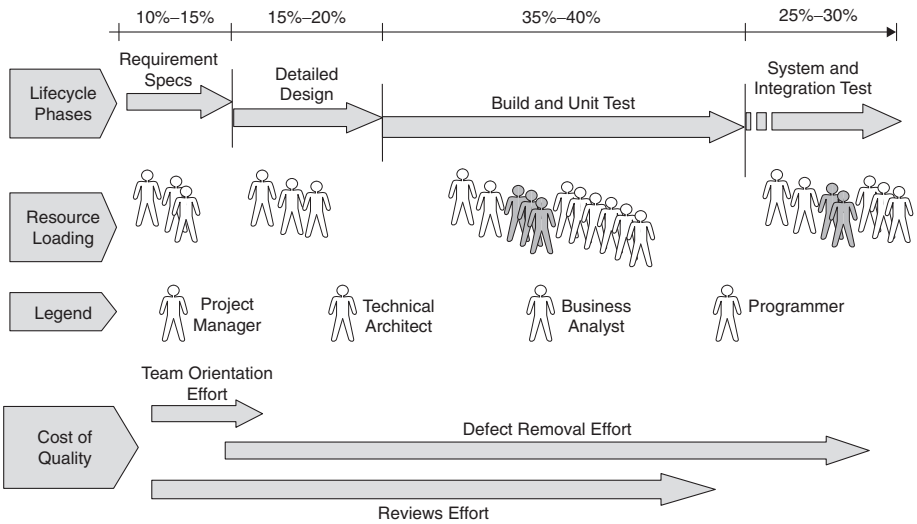

Figure 7.2 *Productivity parameters.*

The number of resources for each category for each lifecycle stage is directly dependent on the project size. Over and above the effort spent by each of the resource categories, a substantial amount of effort goes toward quality and audit-related checks throughout the project execution lifecycle. This quality-related effort is part of the *cost of quality.* **Cost of quality** is the cost an organization spends to attain a certain level of quality output.

Measuring Input

Input is the effort spent by various resources throughout the project execution lifecycle toward various kinds of activities like requirement analysis, high level and detail design (architecture), build and unit test, and system and integration tests. Also included are efforts toward cost of quality activities, as shown in Figure 7.2. The efforts thus can be classified as:

- Requirements/business analysis effort = # business analysts × assigned duration
- Architecture and design effort = # architects/technical leads × assigned duration
- Coding and unit testing effort = # programmers × assigned duration
- System and integration effort = (# testers + # programmers) × assigned duration
- Cost of quality = # reviewers × assigned duration for each + effort spent on defect removal + effort spent on orienting project team

Productivity

Adding all these efforts gives the overall effort for the entire project.

Productivity of the team = application size/total effort

Organization-wide baselines for different technologies are prepared using the preceding methods and are used in future projects. Execution of the future projects is improved through experience. In the absence of specific technology baseline productivity, the technology that is the closest is chosen. For example, if the productivity information for C++ is not available, you can start with the productivity figures of C or Java. If nothing close is found, you can use the weighted average productivity from Capers Jones's table of the languages and databases involved. (Refer to http://www.spr.com for more details on Capers Jones's tables.)

Often IT organizations face a dilemma while defining the list of software project execution activities that are included or excluded while calculating the delivery rate of the project team. There is no defined rule for this. It is best defined by individual organizations, based on the project execution processes they have adopted. Following is an illustrative list of activities that can be included when expressing the productivity for a typical software project:

- Requirement analysis
- High level design
- Detailed design
- Build and unit test
- Integration testing
- System testing
- User testing support
- Product quality checks and reviews

Different organizations adopt different methods of including or excluding certain lifecycle activities. Following is an illustrative list of some of the activities that are typically not considered while calculating productivity:

- Project management effort
- Configuration management effort
- Additional quality assurance efforts towards attaining SEI-CMMI Level 4-5
- User acceptance testing activity
- Developing prototype
- Warranty support
- Holidays and personal leaves

Effort and Schedule

Sizing the project by using function points, SLOC, or other methods is a job only half done. Transforming the size to a deliverable effort within a comfortable schedule makes the project planning a complete success story. Further, the total project effort (for example, in person months) that needs to be consumed in a given schedule provides the guidance to do a proper resource loading.

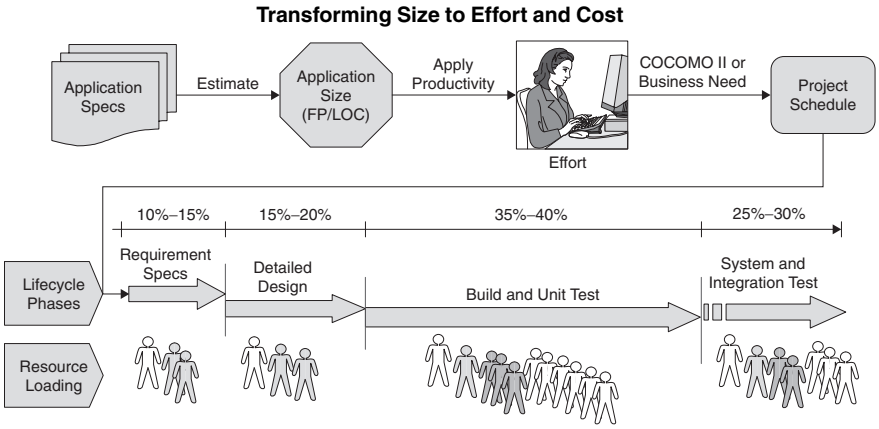

Figure 7.3 *Deriving effort and costs.*

Once the phase-wise resource loading details are available, you can apply the resource rate to each category of resource—such as project manager, architect, analyst, and developer—for the duration of the assignment. Thus the total base cost for the project is calculated. You can then add project management, configuration management, and other overheads as appropriate to get the gross cost. Figure 7.3 shows the broad parameters that are to be taken into account during different lifecycle stages of the project execution.

Deriving Effort

The overall project effort (typically measured in person months) is directly dependent on two critical inputs: application size and project team/programmer productivity. The steps to calculate each of these items are as follows:

- From the given specification for the application, calculate the size of the application. The size can be estimated by using one of the popular estimation methods, such as
  - *Function points method:* Output will be in FP count.
  - *Object points method:* Output will be a list of classes of simple/medium/complex categories.
  - *SLOC method:* Output will be a "gut feel" of lines of code.
- Make sure that you have the productivity (delivery rate) available for the technology platform on which the application is being developed. For every language there are available average productivity figures

that should be adjusted by the historic project productivity data for your own IT organization. Productivity of your project team:
  - Is based on competency of programmers
  - Is specific to a given technology
  - Is dependent on the software development environment
- Convert application size to effort (in person months):
  - Effort = Application size × productivity
- The effort thus derived is the total project effort that would be spent for all the lifecycle stages of the project, from requirements creation through user acceptance. Add project management and configuration management effort as applicable. The effort is also the aggregate of the individual effort spent by each of the resources assigned to the project.

Scheduling

Transforming the overall project effort into a delivery schedule (elapsed time) is somewhat tricky. If the right approach is not applied, the risks of project failure are high. There are three alternatives to calculate the schedule:

- Use popular scheduling methods like COCOMO II.
- "Gut feel" scheduling based on past experience.
- Schedule driven by business user need.

The schedule data that can be obtained by one of these methods is in the form of duration required to deliver the project itself. For example, the schedule could span 10 months from the start date of the project. The schedule thus encompasses all the lifecycle stages of the entire project. From the total duration given to the project team, the project manager must divide the time into lifecycle-based segments. The lifecycle phase percentage is also to be based on historical delivery information of the IT organization. For example, with 10 months of elapsed time, the schedule can be split as follows:

- *Requirements:* 2 months (20 percent)
- *Detailed design:* 1.5 months (15 percent)
- *Build and unit test:* 4 months (40 percent)
- *System and integration test:* 2.5 months (25 percent)

Resource Loading

Resource loading is a complex activity and has to be worked on with extreme care. Improper assignment of resources will have an impact on

Table 7.2 *Resource Loading Chart*

Resource	*M1*	*M2*	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*	*M10*	*Total PM*
Project manager	1	1	1	1	1	1	1	1	1	1	10
Technical analyst		1	1	1					1	1	5
Business analyst	2	3	3	3	3	3	3	3	2	2	27
Programmer			4	4	6	8	10	8	6	4	50
Configuration controller			1	1	1	1	1	1	1	1	8
Total effort	3	5	10	10	11	13	15	14	11	8	100

[M1 = Month 1] [Total PM = Total Person Months].

project delivery schedules as well as the quality of outputs. Resource loading requires two critical mapping considerations:

- The right resource role for the appropriate lifecycle stage. For example, you need to know when to assign a project manager, an architect, or a programmer.
- The right duration of assignment. This includes when to assign and when to release. The effort spent by each resource is determined by tactful resource allocation method.

For Figure 7.3 shown earlier in this section, the resource loading patterns are displayed illustratively in Table 7.2. For your project, you can prepare a table showing resource role assignments for the appropriate durations. For example, assume a total project effort of 100 person months. This effort includes project management and configuration management effort. Table 7.2 illustrates the typical resource loading based on the percentage breakup of elapsed time, as given in the example in this chapter.

NOTE The elapsed time percentage need not be exactly equivalent to resource person months spent in a given lifecycle stage. For example, the requirements phase could be 2 months (20 percent) elapsed time, but the actual resource efforts spent as shown in the table for months M1 and M2 is only 8 (3+5) person months, which is only 8 percent of the total effort for the project. Typically, maximum effort is spent during the build and unit test phase.

Costing

Once the resource loading chart (as shown in Table 7.2) is complete, it is fairly easy to attach the rate per hour (or rate per week/month) for each of the resource roles, such as project manager, architect, analyst, developer, etc. The steps are

- Arrive at the rate per time unit for each of the resources.
- From the resource loading chart, obtain the duration of assignment for each category of resource (project manager, architect, analyst, developer).
- Multiply the individual resource allocation duration by the rate to obtain individual resource costs.
- Aggregate the individual costs to get the overall project cost for resources.
- Add overheads and buffers as applicable.

Conclusion

Sizing development effort using a well-established estimation method like FPA can be a very powerful yardstick you can use to effectively estimate a number of software project execution processes. Through well-defined sizing and costing methods, IT groups within organizations can derive a variety of benefits in various activities that include software contracts, project management, cost of ownership, IT budgets, outsourcing costs, and more. Here are some key areas of direct benefit:

- Because software is "soft," it has always been a difficult product to measure in real terms. FPA-based sizing can provide a reliable yardstick by which to measure software.
- Size and complexity are key, and the core input, to all software estimations. All the subsequent information about software projects, such as effort and schedule based on the skills of the team and cost based on resource rates, can now be better estimated.
- For a CIO, the comfort of being able to compare and equate two different applications (systems) in their IT organization for purposes of TCO (Total Cost of Ownership), budgets, or other strategizing

purposes has always been another area of serious concern. For example, equating an Order Management System with a Payment Processing System on TCO or number of resources deployed would be quite difficult. Even if the two systems belong to the same organization and have been developed on the same technology platform, equating them on any terms would be a difficult task. Sizing the two applications using a common yardstick (like FP count) would perhaps be the nearest to showing their relation accurately.

- Skills or competency of the software development team (productivity) can be better compared through a common yardstick like FP count even if skills are measured across different technology platforms.
- Measuring and monitoring the quality metrics such as effort and schedule variance and also code defect density, etc., can be done by using the sizing technique.
- For a CIO, there are a number of cost-saving, better budgeting, and project monitoring facilities that can be fine-tuned through application sizing methods.

References

1. Ross, Mike. "Size Does Matter: Continuous Size Estimating and Tracking." Quantitative Software Management, Inc.
2. International Function Point Users Group (IFPUG). *Function Point Counting Practices Manual (CPM) Release 4.2.*
3. Peters, Kathleen. "Software Project Estimation." Software Productivity Centre Inc., kpeters@spc.ca
4. Robyn, Lawrie, and Paul Radford. "Using Function Points in Early Life Cycle Estimation." CHARISMATEK Software Metrics.
5. McConnell, Steve. *Rapid Development*. Microsoft Press, 1996.
6. Feature Points, developed by Capers Jones of Software Productivity, Inc., is a variant of IFPUG Function Point Analysis. www.spr.com/products/feature.shtm

Other Interesting Reading Material

McConnell, Steve. "Effort Estimation; Schedule Estimation," in *Rapid Development*. Microsoft Press, 1996. Pp 182–198.

Jones, Capers. *Applied Software Measurement, Second Edition*. McGraw-Hill, 1996.

———. *Software Quality: Analysis and Guidelines for Success*. International Thomson Computer Press, 1997.

———. *Estimating Software Costs*. McGraw-Hill, 1998.