6

The Work Breakdown Structure and Project Estimation

CHAPTER OVERVIEW

Chapter 6 focuses on developing the work breakdown structure, as well as on introducing a number of project estimation approaches, tools, and techniques. After studying this chapter, you should understand and be able to:

- Develop a work breakdown structure.
- Describe the difference between a deliverable and a milestone.
- Describe and apply several project estimation methods. These include the Delphi technique, time boxing, top-down estimation, and bottom-up estimation.
- Describe and apply several softwareœengineering estimation approaches. These include lines of code (LOG), function point analysis, COCOMO, and heuristics.

GLOBAL TECHNOLOGY SOLUTIONS

The white board in the GTS conference room was filled with multicolor markings reflecting the ideas and suggestions from the Husky Air team. Several empty pizza boxes were piled neatly in the corner. It had been an all-day working session for the Husky Air project team. Although it was late in the day, the energy in the room was still high. Everyone felt they were drawing closer to a first draft of the project plan.

Tim Williams stood up and walked over to the electronic white board. Addressing the group, he said, "It looks like we have just about everything we need, but I would like to make sure all of the activities or tasks in the systems testing phase are defined more clearly. Let's start out by identifying what deliverables we need to produce as a result of the testing phase."

Sitaramin paged through his notes and said that the team had identified a test plan and a test results report as part of the project scope. Yan, the project's database administrator, suggested that the test report summarize not only the results of

the system tests, but also what was tested and how the tests were conducted. The rest of the team agreed, and Tim wrote *TESTING PHASE* in capital letters on the board and then *Deliverable: Test Results Report* underneath it. Yan then suggested that the phase needed a milestone. Sitaramin said that the testing phase would not be completed when the report was finished, but only when the test results were acceptable to the client. The rest of the team agreed and Tim wrote *Milestone: Client signs off on test results.*

Tim then asked what specific activities or tasks the team would have to do to create the test results report. For the next ten minutes, the entire team brainstormed ideas. Tim dutifully wrote each idea on the board without judgment and only asked for clarification or help spelling a particular word. After working together for only a short time, the team had already adopted an unwritten rule that no one was to evaluate an idea until after they finished the brainstorming activity. They had found that this encouraged participation from everyone and allowed for more creative ideas.

After a few minutes, the frequency of new ideas suggested by the team started to slow. Tim then asked if any of these ideas or suggestions were similar—i.e., did they have the same meaning or could they be grouped. Again, everyone had ideas and suggestions, and Tim rewrote the original list until the team agreed on a list of activities that would allow them to develop the test results plan.

"This looks pretty good!" exclaimed Tim. Then he added, "But do all of these activities have to be followed one after the other? Or can some of these activities be completed in parallel by different team members?"

Once again, the team began making suggestions and discussing ideas of how to best sequence these activities. This only took a few minutes, but everyone could to see how the testing phase of the project was taking shape. Tim paused, took a few steps back, and announced, "Ok, it looks like we're headed in the right direction. Now who will be responsible for completing these tasks and what resources will they need?"

Since everyone on the team had a specific role, the assigning of team members to the tasks was pretty straightforward. Some of the tasks required only one person, while others needed two or more. The team also identified a few activities where the same person was assigned to tasks scheduled at the same time. The team's discussion also identified an important activity that was overlooked and needed to be added.

Tim joked that he was glad they were using a white board that could easily be erased as he carefully updated the activities and assignments. Then he smiled and said, "Our work breakdown structure is almost complete. All we need to do now is estimate how long each of these testing activities will take. Once we have these estimates, we can enter the work breakdown structure into the project management software package we're using to get the schedule and budget. I think we'll need to review our project plan as a team at least one more time before we present it to our client. I'm sure we'll have to make some changes along the way, but I would say the bulk of our planning work is almost complete."

It was getting late in the day, and the team was starting to get tired. Ted, a telecommunications specialist, suggested that they all meet the next day to finalize the time estimates for the testing phase activities. He also asked that before they adjourned, the team should once again develop an action plan based upon facts the team knew to be true, any assumptions to be tested, and what they would need to find out in order to estimate each of the testing phase activities.

The rest of the team agreed, and they began another learning cycle.

Things to Think About

1. What are some advantages of a project team working together to develop the project plan? What are some disadvantages?
2. Why should the project team members not be too quick to judge the ideas and suggestions provided during a brainstorming session?
3. How can the concept of learning cycles support the project planning process?

INTRODUCTION

In the last chapter, you learned about defining and managing the project's scope, i.e., j the work to be done in order to achieve the project's MOV or goal. Defining and understanding what you have to do is an important first step to determining how you're going to do the work that has to be done. In this chapter, we will focus on defining the tasks or activities that need to be carried out in order to complete all of the scope-related deliverables as promised. Moreover, we also need to estimate or forecast the amount of time each activity will take so that we can determine the overall project schedule.

The Project Management Body of Knowledge (PMBOK) area called project **time management** focuses on the processes necessary to develop the project schedule and to ensure that the project is completed on time. As defined in the PMBOK, project time management includes:

- *Activity definition*—identifying what activities must be completed in order to produce the project scope deliverables.
- *Activity sequencing*—determining whether activities can be completed sequentially or in parallel and any dependencies that may exist among them.
- *Activity duration estimation*—estimating the time to complete each activity.
- *Schedule development*—based upon the availability of resources, the activi ties, their sequence, and time estimates, a schedule for the entire budget can be developed.
- *Schedule control*—ensuring that proper processes and procedures are in place in order to control changes to the project schedule.

In this chapter, we will concentrate on two of these processes: activity definition and activity estimation. These are key processes that deserve special attention because they are required inputs for developing the project network model that will determine the project's schedule and budget. In the next chapter, you will see how we put this all together to develop the detailed project plan.

The remainder of this chapter will introduce several important tools, techniques, and concepts. A **work breakdown structure (WBS)** is discussed first. It provides a hierarchical structure that outlines the activities or work that needs to be done in order to complete the project scope. The WBS also provides a bridge or link between the project's scope and the detailed project plan that will be entered into a project management software package.

Today, most project management software packages are relatively inexpensive and rich in features. It is almost unthinkable that anyone would plan and manage a project without such a tool. Project success, however, will not be determined by one's familiarity with a project management software package or the ability to produce nice

looking reports and graphs. It is the thought process that must be followed before using the tool that counts! Thinking carefully through the activities and their estimated durations first will make the use of a project management software package much more effective. You can still create nice looking reports and graphs, but you'll have more confidence in what those reports and graphs say.

Once the project activities are defined, the next step is to forecast, or estimate, how long each activity will take. Although a number of estimation methods and techniques are introduced here. Estimation is not an exact science. It is dependent upon a number of variables—the complexity of activity, the resources (i.e., people) assigned to complete the activity, and the tools and environment to support those individuals working on the activity (i.e., technology, facilities, etc.). Moreover, confidence in estimates will be lower early in the project because a full understanding of the problem or opportunity at hand is probably lacking. However, as we learn and uncover new information from our involvement in the project, our understanding of the project will increase as well. Although estimates may have to be revised periodically, we should gain more confidence in the updated schedule and budget. Even though no single estimation method will provide 100 percent accuracy all of the time, using one or a combination of methods is preferable to guessing.

THE WORK BREAKDOWN STRUCTURE (WBS)

In the last chapter, you learned how to define and manage the project's scope. As part of the scope definition process, several tools and techniques were introduced. For example, the deliverable definition table (DDT) and deliverable structure chart (DSC) identify the deliverables that must be provided by the project team.

Once the project's scope is defined, the next step is to define the activities or tasks the project team must do to fulfill the scope deliverable requirements. The work breakdown structure (WBS) is a useful tool for developing the project plan and links the project's scope to the schedule and budget. According to Gregory T. Haugan (2002),

> The WBS represents a logical decomposition of the work to be performed and focuses on how the product, service, or result is naturally subdivided. It is an outline of what work is to be performed. (17)

The WBS provides a framework for developing a tactical plan to structure the project work. PMBOK originally defined the WBS as a "deliverable-oriented hierarchy," but much debate and confusion has existed as to what a WBS should look like and how one should be built. Recently, the Project Management Institute formed a committee to recommend standards for the WBS. That committee recommends that no arbitrary limits should be imposed because the WBS should be flexible. Subsequently, the WBS can be used in different ways depending on the needs of the project manager and team.

Work Packages

The WBS decomposes, or subdivides, the project into smaller components and more manageable units of work called work packages. Work packages provide a logical basis for defining the project activities and assigning resources to those activities so that all the project work is identified (Haugan 2002). A work package makes it possible to develop a project plan, schedule, and budget and then later monitor the project's progress.

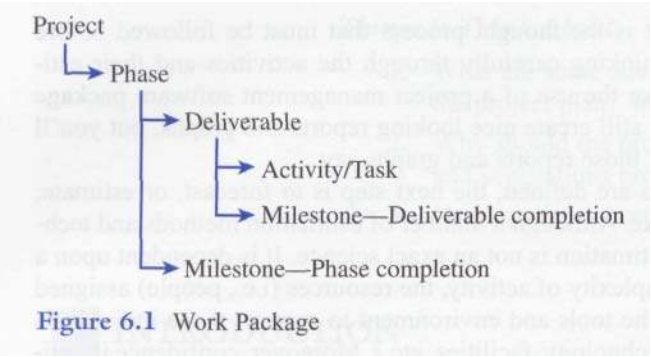

Figure 6.1 Work Package

As illustrated in Figure 6.1, a work package may be viewed as a hierarchy that starts with the project itself. The project is then decomposed into phases, with each phase having one or more deliver-ables as defined in the deliverable definition table and deliverable structure chart. More specifically, each phase should provide at least one specific deliverable—that is, a tangible and verifiable piece of work. Subsequently, activities or tasks are identified in order to produce the project's deliverables.

Deliverables and Milestones

One departure from most traditional views of a WBS is the inclusion of milestones. A **milestone** is a significant event or achievement that provides evidence that that deliverable has been completed or that a phase is formally over.

Deliverables and milestones are closely related, but they are not the same thing. Deliverables can include such things as presentations or reports, plans, prototypes, and the final application system. A milestone, on the other hand, must focus on an achievement. For example, a deliverable may be a prototype of the user interface, but the milestone would be a stakeholder's formal acceptance of the user interface. Only the formal acceptance or approval of the user interface by the project sponsor would allow the project team to move on to the next phase of the project.

In theory, if a project team succeeds in meeting all of its scheduled milestones, then the project should finish as planned. Milestones also provide several other advantages. First, milestones can keep the project team focused. It is much easier to concentrate your attention and efforts on a series of smaller, short-term deliverables than on a single, much larger deliverable scheduled for completion well into the future. On the other hand, if milestones are realistic, they can motivate a project team if their attainment is viewed as a success. If meeting a milestone signifies an important event, then the team should take pleasure in these successes before gearing up for the next milestone.

Milestones also reduce the risk of a project. The passing of a milestone, especially a phase milestone, should provide an opportunity to review the progress of the project. Additional resources should be committed at the successful completion of each milestone, while appropriate plans and steps should be taken if the project cannot meet its milestones.

Milestones can also be used to reduce risk by acting as **cruxes** or proof of concepts. Many times a significant risk associated with IT projects is the dependency on new technology or unique applications of the technology. A crux can be the testing of an idea, concept, or technology that is critical to the project's success. For example, suppose that an organization is building a data warehouse using a particular vendor's relational database product for the first time. A crux for this project may be the collection of data from several different legacy systems, cleansing this data, and then making it available in the relational database management system. The team may ensure that this can be accomplished using only a small amount of test data. Once the project team solves this problem on a smaller scale, they have proof that the concept or technique for importing the data from several legacy systems into the data warehouse can be done successfully. This breakthrough can allow them to incorporate what they have learned on a much larger scale. Subsequently, solving this crux is a

milestone that would encourage the organization to invest more time and resources to complete the project.

Milestones can also provide a mechanism for quality control. Continuing with our example, just providing the users with an interface does not guarantee that it will be acceptable to them. Therefore, the completion of user interface deliverable should end only with their acceptance; otherwise, the team will be forced to make revisions. In short, the deliverable must not only be done, but must be done right.

Developing the WBS

Developing the WBS may require several versions until everyone is comfortable and confident that all of the work activities have been included. It is also a good idea to involve those who will be doing the work—after all, they probably know what has to be done better than anyone else.

The WBS can be quite involved, depending upon the nature and size of the project. To illustrate the steps involved, let's continue with our electronic commerce project example from the last chapter. As you may recall, we created a DDT and DSC to define the scope of the project. To make things easier to follow, let's focus on only one portion of the project—creating a document called the test results report. Figure 6.2 provides the DSC that we developed in Chapter 5. As you can see, two deliver-ables—the test plan and test results report—are to be completed and delivered during the testing phase of the project.

The DSC defines the phases and deliverables for our project. The next step is to develop sets of work packages for each of the phases and deliverables. After a team meeting, let's say that we have identified and discussed several activities that we need to do in order to produce the test results document:

- Review the test plan with the client so that key stakeholders are clear as to what we will be testing, how we will conduct the tests, and when the tests

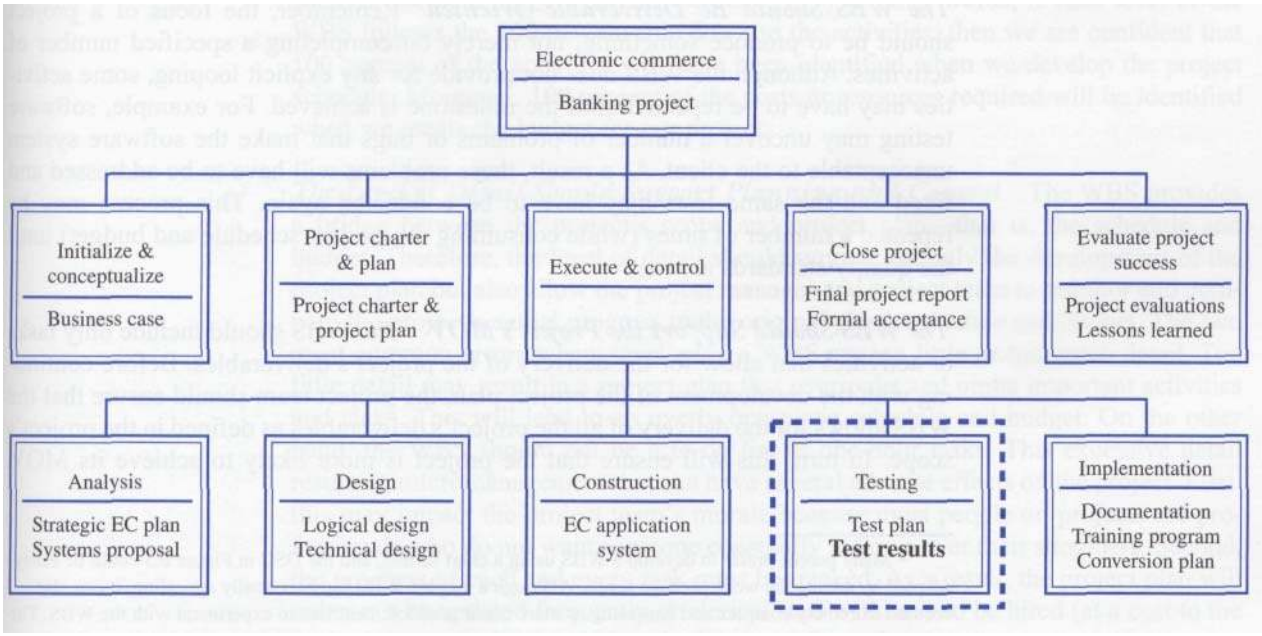

Figure 6.2 Deliverable Structure Chart (DSC) for EC Example

will be carried out. This review may be done as a courtesy or because we need specific support from the client's organization and, therefore, must inform them when that support will be required.

- After we have informed the client that we will test the system, we basically carry out the tests outlined in the test plan.
- *Once we have collected the test results, we need to analyze them.*
- After we analyze the results, we will need to summarize them in the form of a report and presentation to the client.
- If all goes well, then the client will approve or signoff on the test results. Then, we can move on to the implementation phase of our project. If all does not go well, we need to address and fix any problems. Keep in mind, that the test phase is not complete just because we have developed a test plan and created a test report. The client will sign off on the test results only if the system meets certain predetermined quality standards.

Figure 6.3 provides an example of a WBS with the details shown for only the testing phase of the project. As you can see, the WBS implements the concept of a work package for the project, phase, deliverable, task/activity, and milestone components that were illustrated in Figure 6.1. This particular WBS follows an outline format with a commonly used decimal numbering system that allows for continuing levels of detail.[1] If a software package is used to create the WBS, signs in front of each item can either hide or show the details. For example, clicking on "-6.2 Test Results Report" would roll up the details of this work package into "+6.2 Test Results Report". Similarly, clicking on any item with a "+" in front of it would expand that item to show the details associated with it.

The skills to develop a useful WBS generally evolve over time with practice and experience. Everyone, experienced or not, should keep in mind the following points when developing a WBS.

The WBS Should Be Deliverable-Oriented Remember, the focus of a project should be to produce something, not merely on completing a specified number of activities. Although the WBS does not provide for any explicit looping, some activities may have to be repeated until the milestone is achieved. For example, software testing may uncover a number of problems or bugs that make the software system unacceptable to the client. As a result, these problems will have to be addressed and fixed and the same tests may have to be conducted again. This process may be repeated a number of times (while consuming the project schedule and budget) until the quality standards are met.

The WBS Should Support the Project's MOV The WBS should include only tasks or activities that allow for the delivery of the project's deliverables. Before continuing with the development of the project plan, the project team should ensure that the WBS allows for the delivery of all the project's deliverables as defined in the project's scope. In turn, this will ensure that the project is more likely to achieve its MOV.

[1] Many people prefer to develop a WBS using a chart format, and the DSC in Figure 6.3 could be easily adapted by adding the work package levels. Although a graphic WBS can be visually appealing, it can also become extremely complex and confusing as more detail is added. Feel free to experiment with the WBS. The correct form will depend on the situation or your preference.

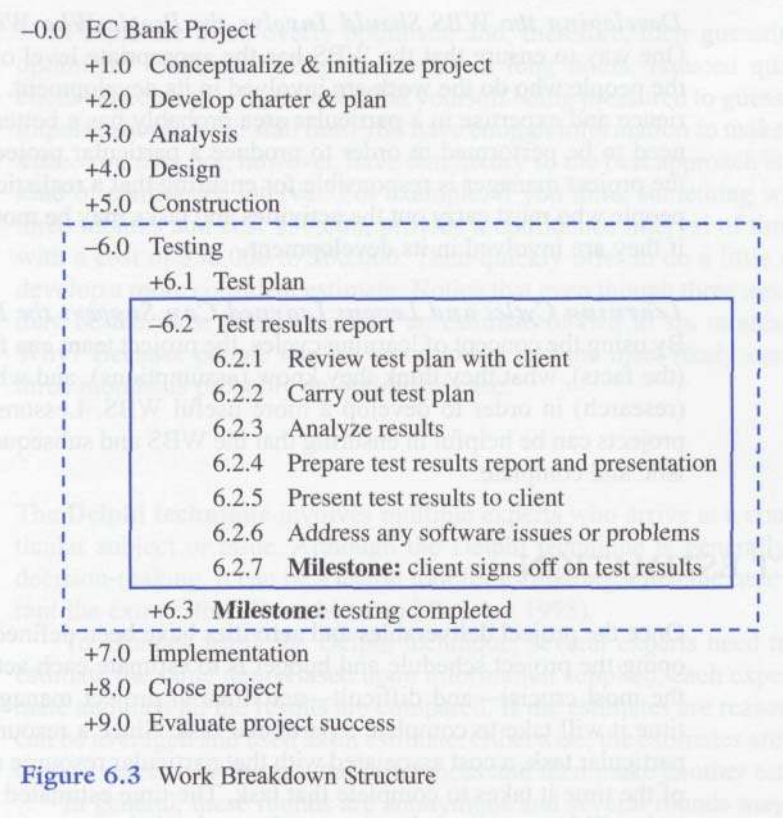

Figure 6.3 Work Breakdown Structure

Haugen (2002) also suggests that the **100 percent rule** is the most important criterion in the developing and evaluating the WBS. The rule states: "The next level decomposition of a WBS element (child level) must represent 100 percent of the work applicable to the next higher (parent) element." (17) In other words, if each level of the WBS follows the 100 percent rule down to the activities, then we are confident that 100 percent of the activities will have been identified when we develop the project schedule. Moreover, 100 percent of the costs or resources required will be identified when we create the budget for our project.

The Level of Detail Should Support Planning and Control The WBS provides a bridge between the project's scope and project plan—that is, the schedule and budget. Therefore, the level of detail should support not only the development of the project plan but also allow the project manager and project team to monitor and compare the project's actual progress to the original plan's schedule and budget. The two most common errors when developing a WBS are too little or too much detail. Too little detail may result in a project plan that overlooks and omits important activities and tasks. This will lead to an overly optimistic schedule and budget. On the other hand, the WBS should not be a to-do list of one-hour tasks. This excessive detail results in micromanagement that can have several adverse effects on the project. First, this may impact the project team's morale because most people on projects are professionals who do not want someone constantly looking over their shoulders. Second, the progress of each and every task must be tracked. As a result, the project plan will either not be updated frequently or clerical staff will have to be hired (at a cost to the project) just to keep everything current.

Developing the WES Should Involve the People Who Will Be Doing the Work One way to ensure that the WBS has the appropriate level of detail is to ensure that the people who do the work are involved in its development. A person who has experience and expertise in a particular area probably has a better feel for what activities need to be performed in order to produce a particular project deliverable. Although the project manager is responsible for ensuring that a realistic WBS is developed, the people who must carry out the activities and tasks may be more committed to the plan if they are involved in its development.

Learning Cycles and Lessons Learned Can Support the Development of a WBS By using the concept of learning cycles, the project team can focus on what they know (the facts), what they think they know (assumptions), and what they need to find out (research) in order to develop a more useful WBS. Lessons learned from previous projects can be helpful in ensuring that the WBS and subsequent project plan are realistic and complete.

PROJECT ESTIMATION

Once the project deliverables and activities have been defined, the next step in developing the project schedule and budget is to estimate each activity's duration. One of the most crucial—and difficult—activities in project management is estimating the time it will take to complete a particular task. Since a resource generally performs a particular task, a cost associated with that particular resource must be allocated as part of the time it takes to complete that task. The time estimated to complete a particular task will have a direct bearing on the project's budget as well. As T. Capers Jones (Jones 1998) points out:

> The seeds of major software disasters are usually sown in the first three months of commencing the software project. Hasty scheduling, irrational commitments, unprofessional estimating techniques, and carelessness of the project management function are the factors that tend to introduce terminal problems. Once a project blindly lurches forward toward an impossible delivery date, the rest of the disaster will occur almost inevitably. (120)

In this section, we will review several estimation techniques—guesstimating, Delphi, top-down and bottom-up estimating.

Guesstimating

Estimation by guessing or just picking numbers out of the air is not the best way to derive a project's schedule and budget. Unfortunately, many inexperienced project managers tend to guesstimate, or guess at the estimates, because it is quick and easy. For example, we might guesstimate that testing will take two weeks. Why two weeks? Why not three weeks? Or ten weeks? Because we are picking numbers out of thin air, the confidence in these estimates will be quite low. You might as well pick numbers out of a hat. The problem is that guessing at the estimates is based on feelings rather than hard evidence.

However, many times a project manager is put on the spot and asked to provide a ballpark figure. Be careful when quoting a time frame or cost off the record, because whatever estimates you come up with often become on the record.

People are often overly optimistic and, therefore, their guesstimates are overly optimistic. Underestimating can result in long hours, reduced quality, and unmet client expectations. If you ever find yourself being pressured to guesstimate, your first impulse should be to stall until you have enough information to make a confident estimate. You may not, however, have that luxury so the best approach is to provide some kind of confidence interval. For example, if you think something will probably take three months and cost $30,000, provide a confidence interval of three to six months with a cost of $30,000 to $60,000. Then quickly offer to do a little more research to develop a more confident estimate. Notice that even though three months and $30,000 may be the most likely estimate, an estimate of two to six months was not made. Why? Because people tend to be optimists and the most likely case of finishing in three months is probably an optimistic case.

Delphi Technique

The Delphi technique involves multiple experts who arrive at a consensus on a particular subject or issue. Although the Delphi technique is generally used for group decision-making, it can be a useful tool for estimating when the time and money warrant the extra effort (Roetzheim and Beasley 1998).

To estimate using the Delphi technique, several experts need to be recruited to estimate the same item. Based upon information supplied, each expert makes an estimate and then all the results are compared. If the estimates are reasonably close, they can be averaged and used as an estimate. Otherwise, the estimates are distributed back to the experts who discuss the differences and then make another estimate.

In general, these rounds are anonymous and several rounds may take place until a consensus is reached. Not surprisingly, using the Delphi technique can take longer and cost more than most estimation methods, but it can be very effective and provide reasonable assurance when the stakes are high and the margin for error is low.

Time Boxing

Time boxing is a technique whereby a *box* of time is allocated for a specific activity or task. This allocation is based more on a requirement rather than on just guesswork. For example, a project team may have two (and only two) weeks to build a prototype. At the end of the two weeks, work on the prototype stops, regardless of whether the prototype is 100 percent complete.

Used effectively, time boxing can help focus the project team's effort on an important and critical task. The schedule pressure to meet a particular deadline, however, may result in long hours and pressure to succeed. Used inappropriately or too often, the project team members become burned out and frustrated.

Top-Down Estimating

Top-down estimating involves estimating the schedule and/or cost of the entire project in terms of how long it *should* take or how much it *should* cost. Top-down estimating is a very common occurrence that often results from a mandate made by upper management (e.g., Thou shalt complete the project within six months and spend no more than $500,000!).

Often the schedule and/or cost estimate is a product of some strategic plan or because someone *thinks* it should take a certain amount of time or cost a particular amount. On the other hand, top-down estimating could be a reaction to the business

environment. For example, the project may have to be completed within six months as a result of a competitor's actions or to win the business of a customer (i.e., the customer needs this in six months).

Once the target objectives in terms of schedule or budget are identified, it is up to the project manager to allocate percentages to the various project life cycle phases and associated tasks or activities. Data from past projects can be very useful in applying percentages and ensuring that the estimates are reasonable. It is important to keep in mind that top-down estimating works well when the target objectives are reasonable, realistic, and achievable.

When made by people independent from the project team, however, these targets are often overly optimistic or overly aggressive. These unrealistic targets often lead to what Ed Yourdon (1999) calls a *death march* project:

> I define a death march project as one whose "project parameters" exceed the norm by at least 50 percent. This doesn't correspond to the "military" definition, and it would be a travesty to compare even the worst software project with the Bataan death march during the Second World War, or the "trail of tears" death march imposed upon Native Americans in the late 1700s. Instead, I use the term as a metaphor, to suggest a "forced march" imposed upon relatively innocent victims, the outcome of which is usually a high casualty rate." (2)

Project parameters include schedule, staff, budget or other resources, and the functionality, features, performance requirements, or other aspects of the project. A death march software project means one or more of the following constraints has been imposed (Yourdon 1999):

- The project schedule has been compressed to less than 50 percent of its original estimate.
- The staff originally assigned or required to complete the project has been reduced to less than 50 percent.
- The budget and resources needed have been reduced by 50 percent or more.
- The functionality, features, or other performance or technical requirements are twice what they should be under typical circumstances.

On the other hand, top-down estimating can be a very effective approach to cost and schedule analysis (Royce 1998). More specifically, a top-down approach may force the project manager to examine the project's risks more closely so that a specific budget or schedule target can be achieved. By understanding the risks, trade-offs, and sensitivities objectively, the various project stakeholders can develop a mutual understanding that leads to better estimation. This outcome, however, requires that all stakeholders be willing to communicate and make trade-offs.

Bottom-Up Estimating

Most real-world estimating is made using **bottom-up estimating** (Royce 1998). Bottom-up estimating involves dividing the project into smaller modules and then directly estimating the time and effort in terms of person-hours, person-weeks, or person-months for each module. The work breakdown structure provides the basis for bottom-up estimating because all of the project phases and activities are defined.

The project manager, or better yet the project team, can provide reasonable time estimates for each activity. In short, bottom-up estimating starts with a list of all

required tasks or activities and then an estimate for the amount of effort is made. The total time and associated cost for each activity provides the basis for the project's target schedule and budget. Although bottom-up estimated is straightforward, confusing effort with progress can be problematic (Brooks 1995).

Continuing with our earlier example, let's assume that after meeting with our software testers, the following durations were estimated for each of the following activities:

6.2	Test results report	
6.2.1	Review test plan with client	1 day
6.2.2	Carry out test plan	5 days
6.2.3	Analyze results	2 days
6.2.4	Prepare test results report and presentation	3 days
6.2.5	Present test results to client	1 day
6.2.6	Address any software issues or problems	5 days

If we add all of the estimated durations together, we find that creating the test results report will take seventeen days. How did we come up with these estimates? Did we guesstimate them? Hopefully not! These estimates could be based on experience—the software testers may have done these activities many times in the past so they know what activities have to be done and how long each activity will take. Or, these estimates could be based on similar or analogous projects. **Analogous estimation** refers to developing estimates based upon one's opinion that there is a significant similarity between the current project and others (Rad 2002).

Keep in mind that estimates are a function of the activity itself, the resources, and the support provided. More specifically, the estimated duration of an activity will first depend upon the nature of the activity in terms of its complexity and degree of structure. In general, highly complex and unstructured activities will take longer to complete than simple, well-structured activities.

The resources assigned to a particular activity will also influence an estimate. For example, assigning an experienced and well-trained individual to a particular task should mean less time is required to complete it than if a novice were assigned. However, experience and expertise are only part of the equation. We also have to consider such things as a person's level of motivation and enthusiasm.

Finally, the support we provide also influences our estimates. Support may include technology, tools, training, and the physical work environment.

These are just some of the variables that we must consider when estimating. You can probably come up with a number of others. Subsequently, estimates will always be a forecast; however, by looking at and understanding the big picture, we can increase our confidence in them.

SOFTWARE ENGINEERING METRICS AND APPROACHES

The discipline of **software engineering** focuses on the processes, tools, and methods for developing a quality approach to developing software (Pressman 2001). **Metrics** on the other hand, provide the basis for software engineering and refers to a broad range of measurements for objectively evaluating computer software.

The greatest challenge for estimating an IT project is estimating the time and effort for the largest deliverable of the project—the application system.

THE MYTHICAL MAN-MONTH

The classic book, *The Mythical Man-Month* by Fredrick P. Brooks, was first published in 1975. Brooks worked at IBM as the manager of a large project that developed the OS/360 operating system. Although the OS/360 was eventually a successful product for IBM, the project was late, took more memory than planned, and cost several times more than originally estimated. In fact, the product did not perform well until after several releases. Based upon his experience, Brooks wrote a number of essays that were embodied in his book. As a result of his timeless advice (and probably due to the fact that some things have not changed, although the term *person-month* may be more appropriate today), a twentieth anniversary edition was issued. The following are some of Brooks' insights:

- "First, our techniques of estimation are poorly developed. More seriously, they reflect an unvoiced assumption which is quite untrue—i.e., that all will go well." (14)
- "Second, our estimating techniques fallaciously confuse effort with progress, hiding the assumption that men and months are interchangeable." (14)
- "Third, because we are uncertain of our estimates, software managers often lack the courteous stubbornness of Antoine's chef (14): Good cooking takes time. If you are made to wait, it is to serve you better, and to please you." (From the menu of Antoine's, a restaurant in New Orleans)
- "Fourth, schedule progress is poorly monitored. Techniques proven and routine in other engineering disciplines are considered radical innovations in software engineering." (14)
- "Fifth, when schedule slippage is recognized, the natural tendency (and traditional) response it to add more manpower. Like dousing a fire with gasoline, this makes matters worse, much worse. More fire requires more gasoline, and thus begins a regenerative cycle which ends in disaster." (14)
- **Brooks Law,** "Adding manpower to a late software project makes it later." (25)

Maintenance projects and the installation of packaged software can experience similar difficulties.

The challenge lies in trying to estimate something that is logical, rather than physical, and that is not well defined until the later stages of the project life cycle. Scope definition can only provide a high-level view of what is and what is not within the scope boundary of the project. Specific requirements, in terms of features and functionality, are generally not defined until later, during the design phase. In addition, the complexity and technical challenges of implementing those features are either unknown or optimistically glossed over in the early stages of the project. As a result, estimating an IT project can be like trying to hit a moving target—hitting either one accurately requires continuous adjustments.

As illustrated in Figure 6.4, the first step to accurately estimating an IT application is determining its size (Jones 1998). In other words, how big is the application? Without getting into too much detail at this point, it should be intuitive that it takes more effort (i.e., in terms of schedule, resources, and budget) to build a larger system than a smaller system. However, the size of the application is only one piece of the estimation puzzle. A good portion of time and effort will be spent on features and functionality that are more complex. As a result, the greater the complexity, the more time and effort that will be spent. Constraints and various influences will also affect the time and effort needed to develop a particular application. These constraints could be attributes of the application (Jones 1998) or include the processes, people, technology, environment, and required quality of the product as well (Royce 1998). Once the resources and time estimates are known, the specific activities or tasks can be sequenced in order to create the project's schedule and budget.

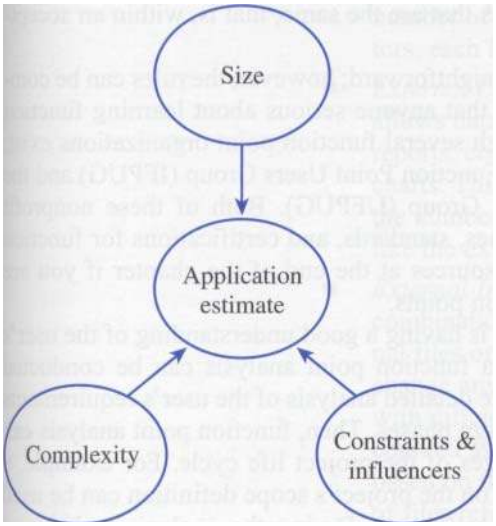

Figure 6.4 Software Engineering Estimation Model

SOURCE: Adapted from Garmus and Herron 1996; Jones 1998, Royce 1998.

Lines of Code (LOC)

Counting the number of lines of code in computer programs is the most traditional and widely used software metric for sizing the application product. It is also the most controversial.

Although counting lines of code seems intuitively obvious—a 1,000 LOC Java program will be ten times larger than a 100 LOC Java program—counting LOC is not all that straightforward. First, what counts as LOC? Do we include comments? Maybe we should not because a programmer could artificially boost his or her productivity by writing one hundred comment lines for every line of code that actually did something. On the other hand, comments are important because they tell us what the code should be doing. This makes it easier to debug and for others to understand what sections of code in the program are doing.

What about declaring variables? Do they count as LOC? In addition, experienced programmers tend to write *less* code than novice programmers. After all, an experienced programmer can write more efficient code, code that does the same thing in fewer lines of code than a novice programmer would use. The same can be said for different programming languages. Writing a program in Assembler requires a great deal more code than writing a similar program in Visual Basic. In fact, one could argue that counting LOC could encourage programmers to write inefficient code, especially when LOC are used as a productivity metric. Finally, it is much easier to count the lines of code after a program is written than it is to estimate how many lines of code will be required to write the program.

Function Points[1]

The inherent problems of LOC as a metric for estimation and productivity necessitated the need for a better software metric. In 1979, Allan Albrecht of IBM proposed the idea of function points at a conference hosted by IBM in Monterey, California (Albrecht 1979). **Function points** are a synthetic metric, similar to ones used every day, such as hours, kilos, tons, nautical miles, degrees Celsius, and so on. However, function points focus on the *functionality* and *complexity* of an application system or a particular module. For example, just as 20 degree Celsius day is warmer than a 10 degree Celsius day, a 1,000 function point application is larger and more complex than a 500 function point application.

The good thing about function points is that they are independent of the technology. More specifically, functionality and the technology are kept separate so we can compare different applications that may or may not use different programming languages or technology platforms. That is, we can compare one application written in COBOL with another application developed in Java. Moreover, function point analysis is reliable—i.e., two people who are skilled and experienced in function point

[1] A more thorough discussion of function point analysis is provided in Appendix A.

analysis will obtain function point counts that are the same, that is, within an acceptable margin of error.

Counting function points is fairly straightforward; however, the rules can be complex for the novice. It is recommended that anyone serious about learning function point analysis become certified. Although several function point organizations exist, the two main ones are the International Function Point Users Group (IFPUG) and the United Kingdom Function Point Users Group (UFPUG). Both of these nonprofit organizations oversee the rules, guidelines, standards, and certifications for function point analysis. In addition, there are resources at the end of the chapter if you are interested in learning more about function points.

The key to counting function points is having a good understanding of the user's requirements. Early on in the project, a function point analysis can be conducted based on the project's scope. Then a more detailed analysis of the user's requirements can be made during the analysis and design phases. Then, function point analysis can and should be conducted at various stages of the project life cycle. For example, a function point analysis conducted based on the project's scope definition can be used for estimation and developing the project's plan. During the analysis and design phases, function points can be used to manage and report progress and for monitoring scope creep. In addition, a function point analysis conducted during or after the project's implementation can be useful for determining whether all of the functionality was delivered. By capturing this information in a repository or database, it can be combined with other metrics useful for benchmarking, estimating future projects, and understanding the impact of new methods, tools, technologies, and best practices that were introduced.

Function point analysis is based on an evaluation of five data and transactional types that define the application boundary as illustrated in Figure 6.5.

- *Internal Logical File (ILF)*—An ILF is a logical file that stores data within the application boundary. For example, each entity in an Entity-Relationship Diagram (ERD) would be considered as an ILF. The complex ity of an ILF can be classified as low, average, or high based on the number of data elements and subgroups of data elements maintained by the ILF. An example of a subgroup would be new customers for an entity called cus tomer. Examples of data elements would be customer number, name, address, phone number, and so forth. In short, ILFs with fewer data ele ments and subgroups will be less complex than ILFs with more data ele ments and subgroups.
- *External Interface File (EIF)*—An EIF is similar to an ILF; however, an EIF is a file maintained by another application system. The complexity of an EIF is determined using the same criteria used for an ILF.
- *External Input (El)*—An El refers to processes or transactional data that originate outside the application and cross the application boundary from outside to inside. The data generally are added, deleted, or updated in one or more files internal to the application (i.e., internal logical files). A com mon example of an El would be a screen that allows the user to input infor mation using a keyboard and a mouse. Data can, however, pass through the application boundary from other applications. For example, a sales system may need a customer's current balance from an accounts receivable system. Based on its complexity, in terms of the number of internal files referenced,

number of data elements (i.e., fields) included, and any other human factors, each El is classified as low, average, or high.

- *External Output (EO)*—Similarly, an EO is a process or transaction that allows data to exit the application boundary. Examples of EOs include reports, confirmation messages, derived or calculated totals, and graphs or charts. This data could go to screens, printers, or other applications. After the number of EOs are counted, they are rated based on their complexity, like the external inputs (El).
- *External Inquiry (EQ)*—An EQ is a process or transaction that includes a combination of inputs and outputs for retrieving data from either the inter nal files or from files external to the application. EQs do not update or change any data stored in a file. They only read this information. Queries with different processing logic or a different input or output format are counted as a single EQ. Once the EQs are identified, they are classified based on their complexity as low, average, or high, according to the number of files referenced and number of data elements included in the query.

Once all of the ILFs, EIFs, Els, EOs, and EQs, are counted and their relative complexities rated, an Unadjusted Function Point (UAF) count is determined. For example, let's say that after reviewing an application system, the following was determined:

- *ILF:* 3 Low, 2 Average, 1 Complex
- *EIF: 2* Average
- *El:* 3 Low, 5 Average, 4 Complex
- *EO:* 4 Low, 2 Average, 1 Complex
- *EQ: 2* Low, 5 Average, 3 Complex

Using Table 6.1, the (UAF) value is calculated.

The next step in function point analysis is to compute a Value Adjustment Factor (VAF). The VAF is based on the Degrees of Influence (DI), often called the Processing Complexity Adjustment (PCA), and is derived from the fourteen General

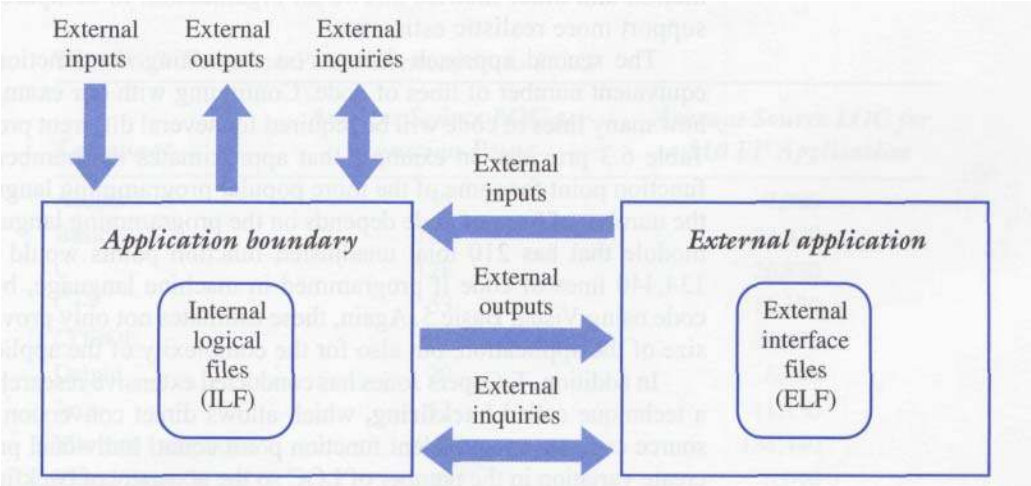

Figure 6.5 The Application Boundary for Function Point Analysis

Table 6.1 Computing UAF

	Complexity			
	Low	*Average*	*High*	*Total*
Internal Logical Files (ILF)	3 × 7 = 21	2 × 10 = 20	1 × 15 = 15	56
External Interface (EIF)	__ × 5 = __	2 × 7 = 14	__ × 10 = __	14
External Input (EI)	3 × 3 = 9	5 × 4 = 20	4 × 6 = 24	53
External Output (EO)	4 × 4 = 16	2 × 5 = 10	1 × 7 = 7	33
External Inquiry (EQ)	2 × 3 = 6	5 × 4 = 20	3 × 6 = 18	44
Total Unadjusted Function Points (UAF)				200

Systems Characteristics (GSC) shown in Table 6.2. To determine the total DI, each GSC is rated based on the following scale from 0 to 5:

- 0 = not present or no influence
- 1 = incidental influence
- 2 = moderate influence
- 3 = average influence
- 4 = significant influence
- 5 = strong influence

Continuing with our example, let's say that after reviewing the application, the degrees of influence shown in Table 6.2 were determined to produce 210 total adjusted function points (TAFP). So what do we do with the total adjusted function point number? Once a total adjusted function point count is calculated, the function point count can be transformed into development estimates. The first approach focuses on productivity—i.e., a person, such as a programmer, can produce a certain number of function points in a given amount of time, such as in a day, a week, or a month. Once again, creating a repository of function point information and other metrics allows an organization to compare various projects and support more realistic estimates.

The second approach focuses on converting the function point count into an equivalent number of lines of code. Continuing with our example, we can determine how many lines of code will be required for several different programming languages. Table 6.3 provides an example that approximates the number of lines of code per function point for some of the more popular programming languages. As you can see, the number of lines of code depends on the programming language. An application or module that has 210 total unadjusted function points would require, for example, 134,440 lines of code if programmed in machine language, but only 6,090 lines of code using Visual Basic 5. Again, these estimates not only provide an estimate for the size of the application, but also for the complexity of the application.

In addition, T. Capers Jones has conducted extensive research and has come up with a technique called **backfiring,** which allows direct conversion from an application's source code to an equivalent function point count. Individual programming styles can create variation in the number of LOG so the accuracy of backfiring is not very high. It can, however, provide an easy way to create a function point inventory of an organization's project portfolio if LOG are readily available.

Table 6.2 GSC and Total Adjusted Function Point

General System Characteristic	*Degree of Influence*
Data communications	3
Distributed data processing	2
Performance	4
Heavily used configuration	3
Transaction rate	3
On-line data entry	4
End user efficiency	4
Online update	3
Complex processing	3
Reusability	2
Installation ease	3
Operational ease	3
Multiple sites	1
Facilitate change	2
Total degrees of influence (TDI)	40
VALUE ADJUSTMENT FACTOR VAF = (TDI * 0.01) + .65	VAF = (40 * .01) + .65 = 1.05
Total adjusted function points = FP = UAF * VAF	FP = 200 * 1.05 = 210

COCOMO

COCOMO is an acronym for Constructive COst MOdel, which was first introduced in 1981 by Barry Boehm in his book *Software Engineering Economics.* Based on LOG estimates, it is used to estimate cost, effort, and schedule (Boehm 1981). The original COCOMO model received widespread interest and is an open model, meaning that all of the underlying equations, assumptions, definitions, and so on are available to the public. The original COCOMO model was based on a study of 63 projects and is a hierarchy of estimation models.

COCOMO is an example of a parametric model because it uses dependent variables, such as cost or duration, based upon one or more independent variables that are quantitative indices of performance and/or physical attributes of the system. Often, parametric models can be refined and fine-tuned for specific projects or projects within specific industries (Rad 2002).

Estimating with COCOMO begins with determining the type of project to be estimated. Project types can be classified as:

- *Organic*—These are routine projects where the technology, processes, and people are expected to all work together smoothly. One may view these types of projects as the easy projects where few problems are expected.

Table 6.3 Function Point Conversion to LOC

Language	*Average Source LOC per Function Point*	*Average Source LOC for a 210 FP Application*
Access	38	7,980
Basic	107	22,470
C	128	26,880
C++	53	11,130
COBOL	107	22,470
Delphi	29	6,090
Java	53	11,130
Machine Language	640	134,440
Visual Basic 5	29	6,090

SOURCE: http://www.spr.com

- *Embedded*—An embedded project is viewed as a challenging project. For example, it may be a system to support a new business process or an area that is new ground for the organization. The people may be less experienced, and the processes and technology may be less mature.
- *Semi-Detached*—If organic projects are viewed as easy and embedded as difficult or challenging, then semi-detached fall somewhere in the middle. These projects may not be simple and straightforward, but the organization feels confident that its processes, people, and technology in place are adequate to meet the challenge.

The basic COCOMO model uses an equation for estimating the number of person-months needed for each of these projects types. A person-month can be thought of as a one-month effort by one person. In COCOMO, a person-month is defined as 152 hours. Once the project type is defined, the level of effort, in terms of person-months, can be determined using the appropriate equation:

- Organic: Person-Months = $2.4 \times \text{KDSI}^{1.05}$
- Semi-Detached: Person-Months = $3.0 \times \text{KDSI}^{1.12}$
- Embedded: Person-Months = $3.6 \times \text{KDSI}^{1.20}$

KDSI = thousands of delivered source instructions, i.e., LOC

Let's suppose that we are developing an application that we estimated to have 200 total adjusted function points. Using Table 6.3, we can convert function points into lines of code. If our application is going to be developed in Java, this would require approximately 10,600 lines of code. If we assume that our project will be of medium difficulty, then the semi-detached equation would be appropriate.

$$\begin{aligned}\text{Person-Months} &= 3.0 \times \text{KDSI}^{1.12} \\ &= 3.0 \times (10.6)^{1.12} \\ &= 42.21\end{aligned}$$

In summary, our 200 function point project will require about 10,600 lines of code and take just over 42.21 person months to complete. Once we have estimated the effort for our project, we can determine how many people will be required. Subsequently, this will determine the time estimate and associated cost for developing our application system.

As Frederick Brooks (1995) points out, people and months are not interchangeable. More people complicate communication and slow things down. Therefore, duration is determined using one of the following formulas:

- Organic: Duration = $2.5 \times \text{Effort}^{0.38}$
- Semi-Detached: Duration = $2.5 \times \text{Effort}^{0.35}$
- Embedded: Duration = $2.5 \times \text{Effort}^{0.32}$

Since our semi-detached project requires 42.21 person-months, the duration of development will be:

$$\begin{aligned}\text{Duration} &= 2.5 \times \text{Effort}^{0.35} \\ &= 2.5 \times (42.21)^{0.35} \\ &= 9.26 \text{ months}\end{aligned}$$

Subsequently, we can determine how many people should be assigned to the development effort:

People Required = Effort ÷ Duration
= 42.21 ÷ 9.26
= 4.55

Therefore, we need 4.55 people working on the project. Okay, so it is pretty tough getting .55 of a person, so we probably will need either four or five people. One could even make an argument that four full-time people and 1 part-time person will be needed for this project.

The above example shows how the basic COCOMO model can be used. There are, however, two other COCOMO models: Intermediate COCOMO and Advanced COCOMO. Intermediate COCOMO estimates the software development effort as a function of size and a set of fifteen subjective cost drivers that include attributes of the end product, the computer used, the personnel staffing, and the project environment. In addition, Advanced COCOMO includes all of the characteristics of Intermediate COCOMO but with an assessment of the cost driver's impact over four phases of development: Product Design, Detailed Design, Coding/Testing, and Integration/Testing.

Today, COCOMO II is available and is more suited for the types of projects being developed using 4GLs or other tools like Visual Basic, Delphi, or Power Builder. However, for more traditional projects using a 3GL, the original COCOMO model can still provide good estimates and is often referred to as COCOMO 81.

Another estimating model that you should be aware of is SLIM, which was developed in the late 1970s by Larry Putnam of Quantitative Software Management (Putnam 1978; Putnam and Fitzsimmons 1979). Like COCOMO, SLIM uses LOC to estimate the project's size and a series of twenty-two questions to calibrate the model.

Heuristics

Heuristics are rules of thumb. Heuristic approaches rely on the fact that the same basic activities will be required for a typical software development project and these activities will require a predictable percentage of the overall effort (Roetzheim and Beasley 1998). For example, when estimating the schedule for a software development task one may, based on previous projects, assign a percentage of the total effort as follows:

- 30 percent Planning
- 20 percent Coding
- 25 percent Component Testing
- 25 percent System Testing

In his book, *Estimating Software Costs*, T. Capers Jones provides a number of heuristics or rules of thumb for estimating software projects based on function points. Some of these rules include:

- Function points raised to the 1.15 power predict approximate page counts for paper documents associated with software projects.
- Creeping user requirements will grow at an average rate of 2 percent per month from the design through coding phases.
- Function points raised to the 1.2 power predict the approximate number of test cases created.

- Function points raised to the 1.25 power predict the approximate defect potential for new software projects.
- Each software test step will find and remove 30 percent of the bugs that are present.
- Each formal design inspection will find and remove 65 percent of the bugs present.
- Each formal code inspection will find and remove 60 percent of the bugs present.
- Maintenance programmers can repair eight bugs per staff month.
- Function points raised to the 0.4 power predict the approximate development schedule in calendar months.
- Function points divided by 150 predict the approximate number of personnel required for the application.
- Function points divided by 750 predict the approximate number of maintenance personnel required to keep the application updated.
- Multiply software development schedules by the number of personnel to predict the approximate number of staff months of effort.

Jones makes an important observation: Rules of thumb are easy, but they are not accurate. As Garmus and Herron point out (Garmus and Herron 1996):

> Accurate estimating is a function of applying a process and recognizing that effort must be expended in creating a baseline of experience that will allow for increased accuracy of that process. Estimating does not require a crystal ball; it simply requires commitment. (142)

Automated Estimating Tools

A number of automated tools can be used for cost, schedule, and resource estimation. These tools include spreadsheets, project management tools, database management systems, software cost estimating, and process or methodology tools. Many of these tools not only help estimate, but also allow the organization to create a database or repository of past projects. In fact, it was found that estimates usually have an accuracy of between 5 and 10 percent when historical data was accurate. Moreover, automated estimating tools are generally more conservative when they are not accurate, as opposed to manual methods that are generally optimistic (Jones 1998).

As the complexity of software development projects increases, the market for software estimation tools will increase as well. Some of the automated tools available include COCOMO II, SLIM, CHECKPOINT, Knowledge Plan, and Cost*Xpert. Research suggests that projects that use a formal estimating tool have a better chance of delivering a system that is on time and within budget.

WHAT IS THE BEST WAY TO ESTIMATE IT PROJECTS?

Unfortunately, no single method or tool is best for accurately estimating IT projects. It may be a good idea to use more than one technique for estimating. You will, however, very likely have two different estimates.

If the estimates from different estimating techniques are fairly close, then you can average them with a fairly high degree of confidence. If the estimates vary widely,

then you should probably be skeptical of one or both estimates and review the data that was collected (Roetzheim and Beasley 1998).

Your initial estimates probably will have to be adjusted up or down based on past experience or data from past projects. Many times, however, the initial estimates are negotiated by upper management or the client. For example, you may come up with an estimate that the project will take twelve months and cost $1.2 million. Unless you can substantiate your estimates, upper management may counter and mandate that the project be completed in eight months and cost no more than $750,000. This counter may be a result of a real business need (i.e., they really do need it in eight months and can not spend more than $750,000) or their belief that you inflated the schedule and budget and some of the fat can be trimmed from your estimates. As a result, you may end up working on a death march project.

It basically comes down to whether the project can or cannot be delivered earlier. It is up to the project manager not only to arrive at an estimate, but also to support the estimates. Otherwise, the project's schedule and budget can be very unrealistic. Working long hours and under intense pressure will surely have a negative impact on the project team. A project manager's team must always come first, and protecting them by having a realistic deadline and adequate resources as defined by the project's schedule and budget is the first step.

CHAPTER SUMMARY

Although defining a project's scope in terms of project-oriented and product-oriented deliverables provides an idea of what must be done, the project manager and team must still develop a tactical approach that determines what needs to be done, when it will be done, who will do the work, and how long will it take. The work breakdown structure (WBS) is an important and useful tool for bridging the project's scope with the detailed project plan. More specifically, the WBS provides a logical hierarchy that decomposes the project scope into work packages. Work packages focus on a particular deliverable and include the activities required to produce the deliverable. In addition, milestones provide a mechanism for ensuring that project work is not only done, but also done right.

Once the work packages have been identified, projected durations must be made. Instead of guesstimat-ing, or guessing at the estimates, a number of project estimation methods and techniques were introduced. Traditional approaches to estimating include:

- *The Delphi Technique*—This approach involves multiple experts who arrive at a consensus after a series of round-robin sessions in which information and opinions are anonymously provided to each expert.
- *Time-Boxing*—A technique where a *box* of time is allocated to a specific task. For example, a team may be given two weeks (and only two weeks) to develop a prototype of a user interface.
- *Top-Down Estimating*—This system involves estimating a schedule or budget based upon how long the project or an activity should take or how much it should cost. For example, the project manager may be told that the project must be completed in six months. The project manager then schedules or estimates the project and activities backwards so that the total duration of the activities adds up to six months or less. Although this approach may be used when competitive necessity is an issue, unrealistic expectations can lead to projects with very little chance of meeting their objectives.
- *Bottom-Up Estimating*—Most real-world estimating uses this approach. The WBS outlines the activities that must be completed, and an estimate is made for each of the activities. The various durations are then added together to determine the total duration of the project. Estimates may be analogous to other projects or based on previous experience. These estimates are also a function of the activity itself (e.g., degree of complexity, structuredness, etc.), the resources assigned (e.g., a person's knowledge, expertise, enthusiasm, etc.) and support (e.g., technology, tools, work environment, etc.).

In addition, several software engineering approaches were introduced for estimating the software development effort. These included:

- *Lines of Code (LOG)*—Although counting or trying to estimate the amount of code that must be written may appear intuitively pleasing, there are a number

of deficiencies with this approach. The number of LOG may provide an idea of the size of a project, but it does not consider the complexity, constraints, or influencers that must be taken into account. *Function Points*—Function points were introduced by Allen Albrecht of IBM in 1979. They are synthetic measures that take into account the functionality and complexity of software. Because function points are independent of the technology or programming language used, one application system can be compared with another. *COCOMO*—The Constructive COst MOdel was introduced by Barry Boehm in 1981. Estimates for a software systems effort are determined by an equation based upon the project's complexity. More specifically, a software project may be classified as organic (relatively simple and straightforward), embedded (difficult), or semi-detached (somewhere in the middle). Once the effort, in terms of person-months, is calculated, a similar procedure using another model can estimate the project's duration. • *Heuristics*—Heuristics are rules of thumb that are applied to estimating a software project. The basic premise is that the same activities will be repeated on most projects. This approach may include assigning a specific percentage of the project schedule to specific activities or using other metrics such as function points.

Estimating the effort and duration of an IT project is not an exact science. No single method or technique will provide 100 percent accuracy. Using a combination of approaches may help triangulate an estimate, which provides a confidence greater than when merely guessing or using a single estimation technique. To be realistic, estimates should be revised as understanding of the project increases and new information acquired.

WEB SITES TO VISIT

www.softwaremetrics.com: Articles and examples for learning more about function point analysis

www.spr.com: The site for Software Productivity Research. Capers Jones articles and information about software estimation and planning tools for IT projects

www.ifpug.org: International Function Point Users Group

sunset.usc.edu/research/COCOMOII/index.html: The latest version and information about COCOMO

REVIEW QUESTIONS

1. Describe the PMBOK area of project time management.
2. What is a WBS? What purpose does it serve?
3. Discuss why a project's scope must be tied to the WBS.
4. What is a work package?
5. What is the difference between a deliverable and a milestone?
6. What purpose do milestones serve?
7. What are some advantages of including milestones in the WBS?
8. What is a crux? Why should the project manager and project team identify the cruxes of a project?
9. What is the proper level of detail for a WBS?
10. Why should the WBS be deliverable-oriented?
11. Explain why people who do the work on a project should be involved in developing the project plan?
12. How does the concept of knowledge management support the development of the project plan?
13. How is estimating an IT project different from estimating a construction project?
14. What makes estimating an IT project challenging?
15. What is guesstimating? Why should a project manager not rely on this technique for estimating a project?
16. Describe the potential problems associated with providing an off-the-record estimate?
17. What is the Delphi technique? When would it be an appropriate estimating technique for an IT project?
18. What is time boxing? What are some advantages and disadvantages of time boxing project activities?
19. Describe top-down estimating. What are some advantages and disadvantages of top-down estimating?
20. Describe bottom-up estimating. What are some advantages and disadvantages of bottom-up estimating?
21. What is a death march project? What situations in project planning can lead to a death march project?

22. Discuss why adding people to a project that is already behind schedule can make it later?
23. What is software engineering?
24. Why is counting lines of code (LOG) a popular method for estimating and tracking programmer productivity? What are some problems associated with this method?
25. What is a function point? What advantages do function points have over counting lines of code?
26. How can function point analysis be used to help manage scope creep?
27. What is backfiring? How could an organization use backfiring to improve the accuracy of estimating IT projects?
28. What is COCOMO?
29. Under the COCOMO model, describe the organic, semi-detached, and embedded models.
30. What are heuristics? Discuss some of the advantages and disadvantages of using heuristics for estimating IT projects.
31. What can lead to inaccurate estimates? How can an organization improve the accuracy of estimating IT projects?
32. What is the impact of consistently estimating too low? Too high?

EXTEND YOUR KNOWLEDGE

1. Develop a deliverable-oriented WBS for a surprise birthday party for a friend or relative (perhaps even your instructor?). Be sure to define a measure of success for this party and include milestones.
2. Using the following phases as a guide, develop a WBS for an IT project that will allow Husky Air to keep track of all scheduled maintenance for its chartered aircraft. For each phase, define a deliverable, several activities or tasks, and a milestone.

   1.0 Conceptualize and Initialize Project
   2.0 Develop Project Charter and Plan
   3.0 Analysis
   4.0 Design
   5.0 Construction
   6.0 Testing
   7.0 Implementation
   8.0 Close Project
   9.0 Evaluate Project Success

3. Using the information below, complete a function point analysis in order to use the basic COCOMO model to estimate the duration and number of people needed to develop an application using C++. Assume that the project is relatively simple and straightforward and that the project team is familiar with both the problem and technology. You can perform the calculations by hand, but feel free to use an appropriate software tool.

	Complexity			
	Low	*Average*	*High*	*Total*
Internal logical files (ILF)	__ × 7 = __	__ × 10 = __	__ × 15 = __	
External interface (EIF)	__ × 5 = __	__ × 7 = __	__ × 10 = __	
External input (EI)	__ × 3 = __	__ × 4 = __	__ × 6 = __	
External output (EO)	__ × 4 = __	__ × 5 = __	__ × 7 = __	
External inquiry (EQ)	__ × 3 = __	__ × 4 = __	__ × 6 = __	

	Complexity		
	Low	*Average*	*High*
Internal logical files (ILF)	4	2	0
External interface (EIF)	0	1	0
External input (EI)	3	2	0
External output (EO)	5	7	3
External inquiry (EQ)	2	5	2

General System Characteristic	*Degree of Influence*
Data communications	2
Distributed data processing	3
Performance	3
Heavily used configuration	4
Transaction rate	4
On-line data entry	2
End user efficiency	2
Online update	2
Complex processing	2
Reusability	3
Installation ease	2
Operational ease	2
Multiple sites	1
Facilitate change	1

Language	*Average Source LOC per Function Point*
Basic	107
C	128
C++	53
COBOL	107
Delphi	29
Java	53
Visual Basic 5	29

BIBLIOGRAPHY

Albrecht, Allan J. 1979. *Measuring Application Development Productivity*. Proceedings SHARE/GUIDE IBM Applications Development Symposium, Monterey, Calif., October 14—17, 1979.

Brooks, F. P. 1995. *The Mythical Man-Month.* Reading, Mass.: Addison Wesley. Boehm, B. W. 1981. *Software Engineering Economics.* Englewood Cliffs, N.J.: Prentice Hall. Brooks, F. P. 1995. *The Mythical Man-Month.* Reading, Mass.: Addison Wesley. Garmus, D. and D. Herron. 1996. *Measuring the Software Process.* Upper Saddle River, N.J.: Prentice Hall PTR. Haugan, G. T. 2002. *Efffective Work Breakdown Structures.* Vienna, Va.: Management Concepts, Inc. Jones, T. C. 1998. *Estimating Software Costs.* New York: McGraw-Hill.

Pressman, R. S. 2001. *Software Engineering: A Practitioner's Approach.* Boston: McGraw-Hill. Putnam, L. H. 1978. General Empirical Solution to the Macro Software Sizing and Estimating Problem. *IEEE Transactions Software Engineering* SE 4(4): 345-361. Putnam, L. H. and A. Fitzsimmons. 1979. Estimating Software Costs. *Datamation* 25(Sept-Nov): 10-12. Rad, P. F. 2002. *Project Estimating and Cost Management.* Vienna, Va.: Management Concepts, Inc. Roetzheim, W. H. and R. A. Beasley. 1998. *Software Project Cost and Schedule Estimating: Best Practices.* Upper Saddle River, N.J.: Prentice Hall. Royce, W. 1998. *Software Project Management: A Unified Framework.* Reading, Mass.: Addison Wesley. Yourdon, E. 1999. *Death March.* Upper Saddle River, N.J.: Prentice Hall.