Chapter4.
Software Project Planning

4.1 Introduction

- Software project planning involves deciding what tasks need to be done, in what order to do the tasks, and what resources are needed to accomplish the tasks.

4.2 WBS—Work Breakdown Structure

- One of the first tasks is to break the large tasks into small tasks. It means finding identifiable parts of the tasks.
- The work breakdown structure (WBS) should be a tree structure. The top-level breakdown usually matches the life cycle model (LCM) used in the organization. The next-level breakdown can match the processes in the organization's process model (PM). Further levels are used to partition the task into smaller, more manageable tasks.

4.2 WBS—Work Breakdown Structure

- The following are rules for constructing a proper work breakdown structure:
  1. The WBS must be a tree structure. There should be no loops or cycles in the WBS. Iterative actions will be shown in the process model and/or the life cycle model.
  2. Every task and deliverable description must be understandable and unambiguous. The purpose of a WBS is communication with team members. If the team members misinterpret what the task or deliverable is supposed to be, there will be problems.
  3. Every task must have a completion criterion (often a deliverable). There must be a way to decide when a task is completed, because subtasks that have no definite ending encourage false expectations of progress. This decision is called a completion criterion. It may be a deliverable, for example, a complete design for the project, and then a peer review can decide if it is complete.
  4. All deliverables (artifacts) must be identified. A deliverable must be produced by some task or it won't be produced.
  5. Positive completion of the tasks must imply completion of the whole task. The purpose of the work breakdown schedule is to identify the subtasks necessary to complete the whole task. If important tasks or deliverables are missing, the whole task will not be accomplished.

4.2 WBS—Work Breakdown Structure

EXAMPLE 4.1

In making a loaf of bread, the life cycle model for cooking might be as shown in Fig. 4-1.

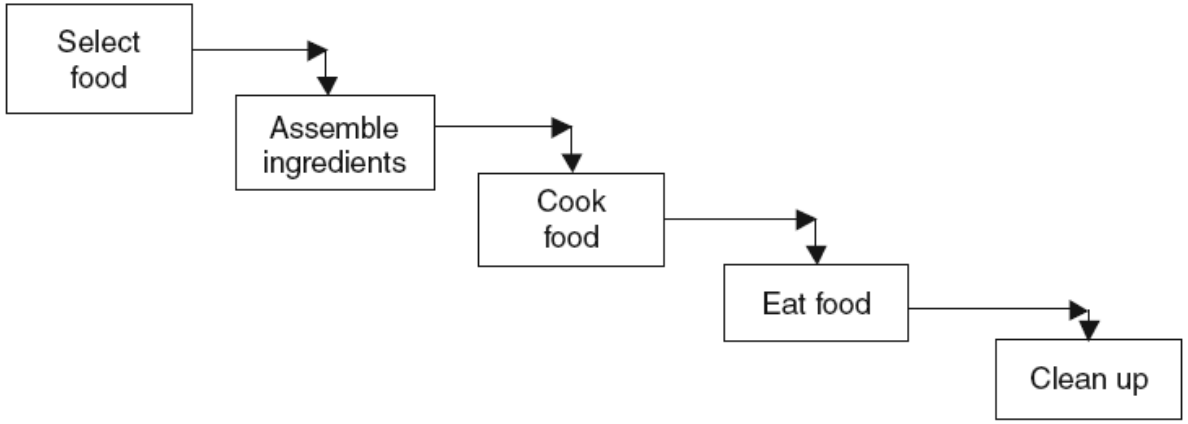


Fig. 4-1. Life cycle model for cooking.

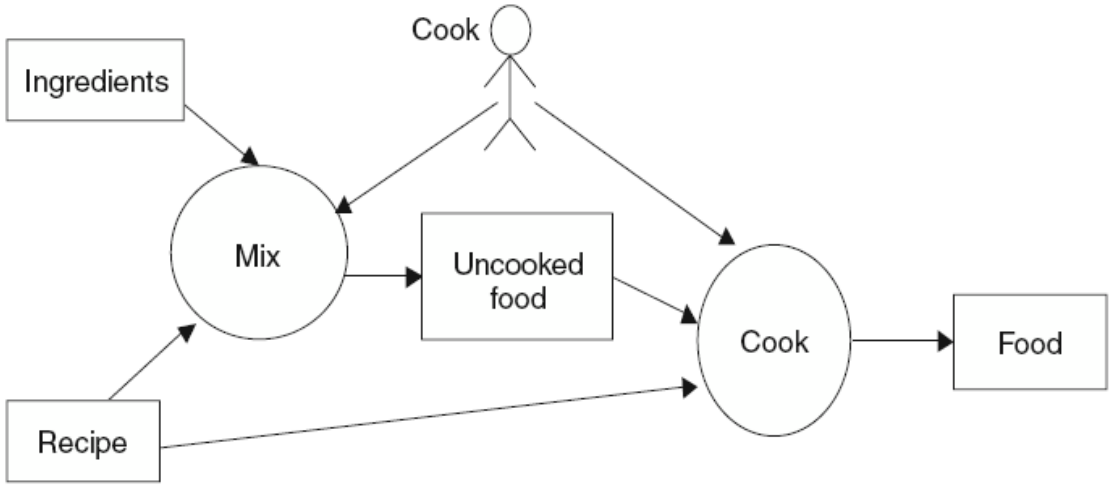


Fig. 4-2. Cooking process model.

The process model for cooking might be as shown in Fig. 4-2.

4.2 WBS—Work Breakdown Structure

The subtasks might be as follows:

Choose ingredients, Check on ingredients, Assemble ingredients, Add liquids, Add yeast, Add small amount of flour, Make sponge (yeast and liquids), Let rise first time, Add remaining flour, Knead, Let rise second time, Form into loaves, Let rise third time, Bake, Slice, Butter, Eat, and Clean up.

These can be divided into the phases of the life cycle model and processes of the process model (the leftmost are LCMs, those with one indent are PMs, those with two indents are tasks, and the deliverables are on the right):

Select food	
Choose ingredients	List of ingredients
Check on ingredients	Shopping list
Assemble ingredients	
Assemble ingredients	Assembled ingredients
Cook food	
Mix	
Add liquids	Liquid in bowl
Add yeast	Liquids with yeast
Add small amount of flour	Liquids and flour
Make sponge (yeast and liquids)	Sponge
Let rise first time	Risen sponge
Add remaining flour and knead	Kneaded dough
Let rise second time	Risen dough
Form into loaves	Loaves
Let rise third time	Risen loaves
Cook	
Bake	Bread
Eat	
Slice	Slice of bread
Butter	Buttered slice
Eat	Good taste
Clean up	
Clean up	Clean kitchen

4.2 WBS—Work Breakdown Structure

EXAMPLE 4.2

Team XYZ wants to develop a face recognition system for use on the robot. The system is intended to greet visitors to the robotics laboratory. It should recognize faces it has seen before with a reasonable reliability. The first pass on the work breakdown might recognize the following subtasks:

Feasibility
- Determine feasibility of vision.
- Determine camera and software availability.
- Schedule camera and vision software acquisition.

Risk Analysis
- Determine vision risks.

Requirements
- Specify requirements.

Design
- Design prototypes.
- Prototype vision.

Implementation
- Code the image capture.
- Code the image processing.
- Code the image comparison.
- Integrate with other robot software.

Testing
- Test image capture.

Delivery
- Document.

4.2 WBS—Work Breakdown Structure

EXAMPLE 4.3

The team broke the subtask ‘‘Code the image capture’’ into a more detailed set of subtasks, with each new subtask having a more specific deliverable and completion criterion. The set of subtasks and deliverables are the following:

Install commercial camera driver.	Installed driver
Test driver from windows and save an image to file.	Image file
Write routine to call driver from C++.	Routine
Test C++ routine separately and save an image to file.	Image from C++ code
Test C++ routine from the main robot control software and capture image.	Image from main

Project Scheduling

- Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed.
- You estimate the calendar time needed to complete each task, the effort required and who will work on the tasks that have been identified.
- You also have to estimate the resources needed to complete each task, such as the disk space required on a server, the time required on specialized hardware, such as a simulator, and what the travel budget will be.

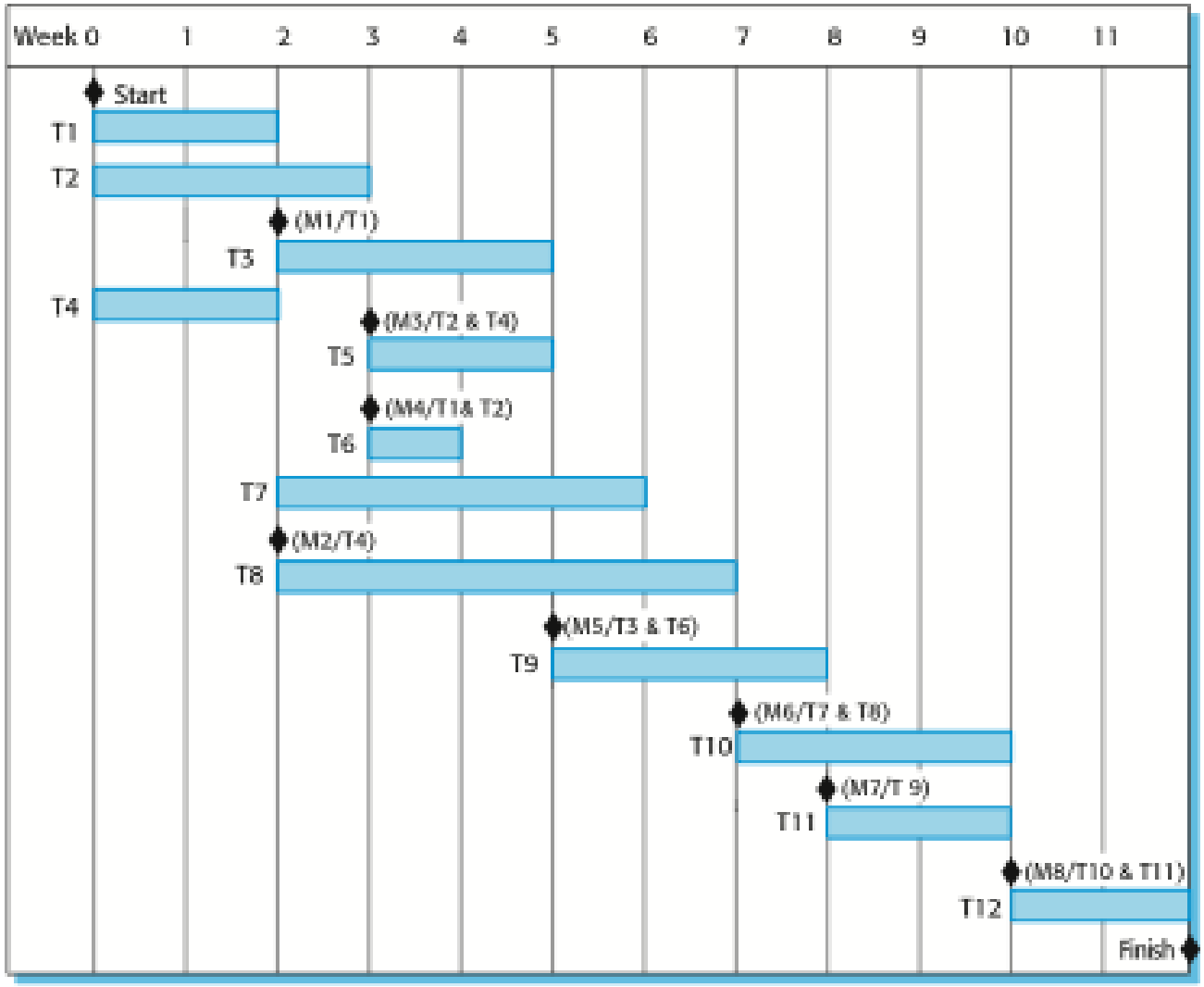
Scheduling Problems

- Estimating the difficulty of problems and hence the cost of developing a solution is hard.
- Productivity is not proportional to the number of people working on a task.
- Adding people to a late project makes it later because of communication overheads.
- The unexpected always happens. Always allow contingency in planning.

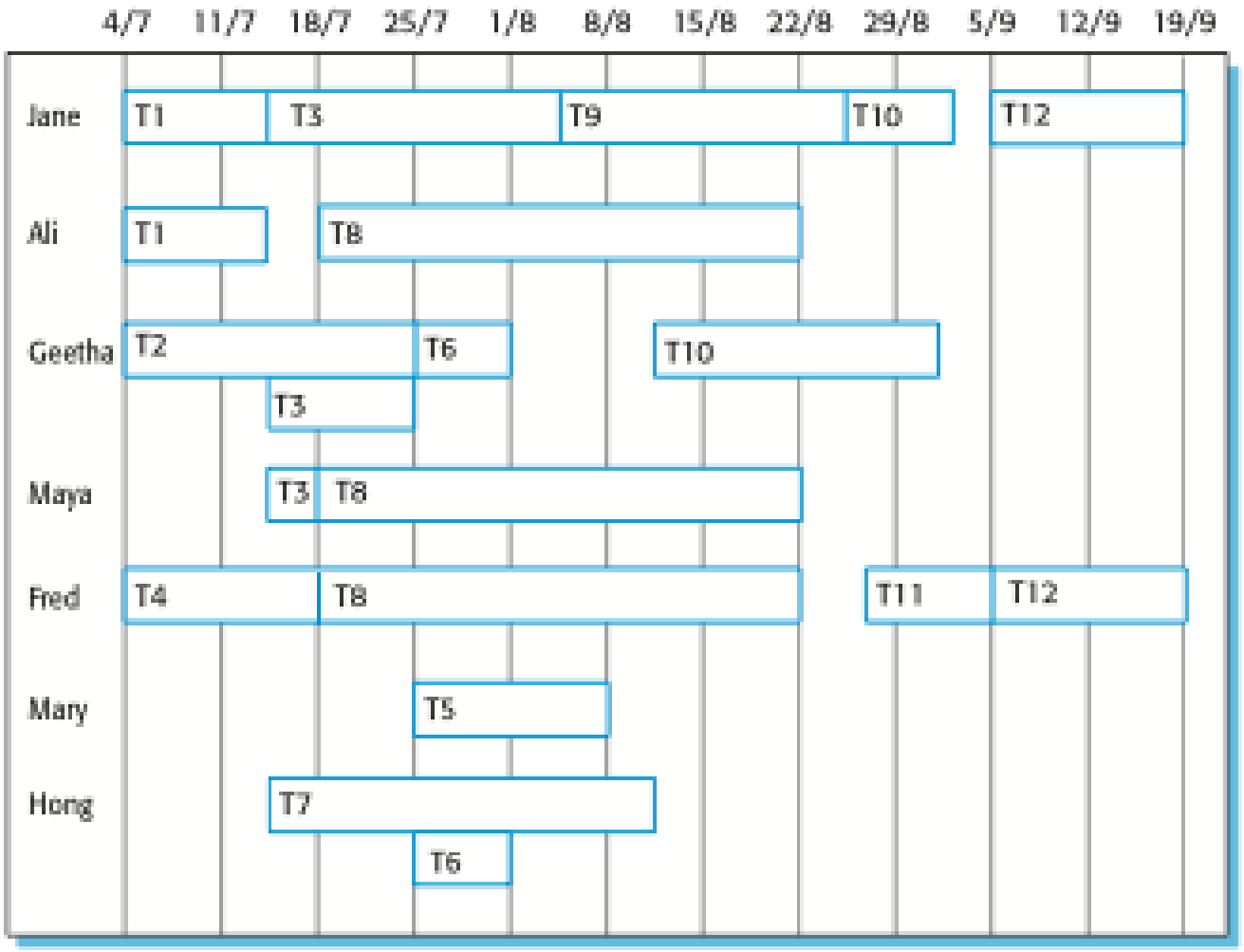
Schedule Representation

- Graphical notations are normally used to illustrate the project schedule.
- These show the project breakdown into tasks. Tasks should not be too small. They should take about a week or two.
- Bar charts are the most commonly used representation for project schedules. They show the schedule as activities or resources against time.

Activity Bar Chart



Staff Allocation Chart



4.3 PERT—Program Evaluation and Review Technique

- This technique creates a graph that shows the dependencies among the tasks. Each task has an estimate of the time necessary to complete the task and a list of other tasks that have to be completed before this task can be started (dependencies).
- The whole task is only completed when all the subtasks are completed. The graph can be used to calculate the completion times for all the subtasks, the minimum completion time for the whole task, and the critical path of the subtasks.

ALGORITHM FOR COMPLETION TIMES

1. For each node, do step 1.1 (until completion times of all nodes are calculated)
   1.1 If the predecessors are completed, then take the latest completions time of the predecessors and add required time for this node.
2. The node with the latest completion time determines the earliest completion time for project.

4.3 PERT—Program Evaluation and Review Technique

EXAMPLE 4.4

Apply this algorithm to Table 4-1, which shows an example of tasks and dependencies. The same dependencies are shown in Fig. 4-3. To apply the completion time algorithm, start with subtask a ; it has no dependencies, so it can start at the initial time (say, 0). It can complete at time 0 + 8 = 8. Similarly, subtask b can complete at time 0 + 10 = 10. See Table 4-2. Note that since these subtasks are not dependent on each other or on anything else, they can start at time 0. Their completion times are calculated without concern for lack of resources. That is, for this completion calculation, it assumes that there are people available to do both tasks at the same time.

Table 4-1 Subtasks

Subtask ID	Time to Complete Task	Dependencies
a	8	
b	10	
c	8	a, b
d	9	a
e	5	b
f	3	c, d
g	2	d, e
h	4	f, g
i	3	e, f

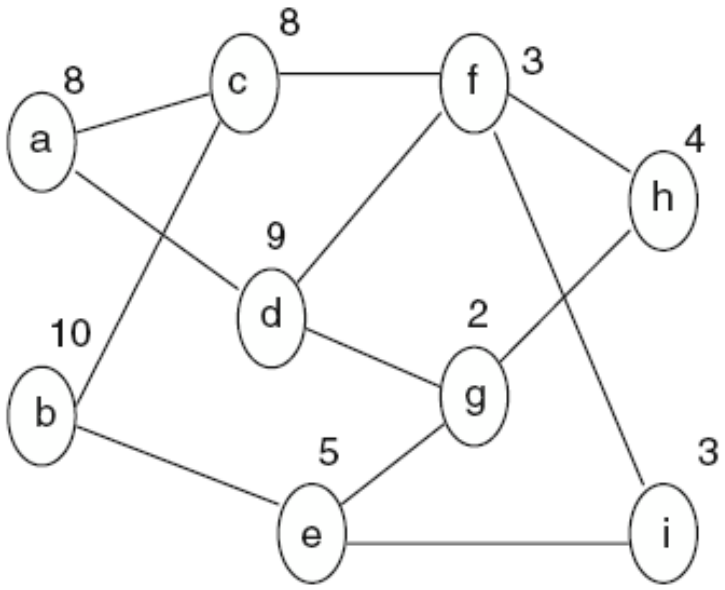


Fig. 4-3. PERT diagram.

4.3 PERT—Program Evaluation and Review Technique

Since the completion times for subtasks a and b are now calculated, the completion times for nodes c, d, and e can be calculated. Since the predecessors of c finish at 8 and 10, subtask c can start at 10 and complete at 10 + 8 = 16. The start time for d will be 8 and the completion time can be 8 + 9 = 17, and for e the times will be 10 and 10 + 5 = 14.

Now we can process subtasks f and g. The start times can be 17 and 16, respectively. The completion times will be 17 + 3 = 20 for f and 16 + 2 = 18 for g. Subtasks h and i can now be calculated with both starting at 21 and h competing at 25 and i at 24. Table 4-2 has all of the start and completion times.

Table 4-2

Subtask ID	Start Time	Completion Time	Critical Path
a	0	8	
b	0	10	*
c	10	18	*
d	8	17	
e	10	15	
f	18	21	*
g	17	19	
h	21	25	*
i	21	24	

4.3 PERT—Program Evaluation and Review Technique

CRITICAL PATH

The critical path is the set of tasks that determines the shortest possible completion time.

ALGORITHMFOR MARKING CRITICAL PATH

1. Start with the node(s) with the latest completion time(s); mark it (them) as critical.
2. Select the predecessor(s) of the critical node(s) with latest completion time(s); mark it (them) as critical. Continue Step 2 until reaching the starting node(s).

4.3 PERT—Program Evaluation and Review Technique

EXAMPLE 4.5

In Table 4-2 we can see the completion times of all of the subtasks. Subtask h has the latest completion time, 25. Thus, we mark h as part of the critical path. The predecessors of h are f and g. Subtask f has the latest completion time of those two subtasks, so f is marked as part of the critical path.

Subtask f has c and d as predecessors. Since c has the later completion time, c is marked as part of the critical path. Subtask c has a and b as predecessors, and since b has the later time, it is part of the critical path. Since we are now at an initial subtask, the critical path is complete.

4.3 PERT—Program Evaluation and Review Technique

SLACK TIME

Subtasks that are on the critical path have to be started as early as possible or else the whole project will be delayed. However, subtasks that are not on the critical path have some flexibility on when they are started. This flexibility is called the slack time.

ALGORITHMFOR SLACK TIME

1. Pick the noncritical node with the latest ending time that has not been processed. If the subtask has no successors, pick the latest ending time of all nodes. If the subtask has successors, pick the earliest of the latest start times of the successor nodes. This is the latest completion time for this subtask. Make the latest start time for this subtask to reflect this time.
2. Repeat Step 1 until all noncritical path subtasks have been processed.

4.3 PERT—Program Evaluation and Review Technique

EXAMPLE 4.6

The noncritical subtask with the latest completion time is subtask i. Since it has no successors, the latest completion time, 25, is used. This is added as the latest completion time for i. Since 25 is 1 later than 24, the start time is changed from 21 to 21,22. Now the latest nonprocessed, noncritical subtask is g. Since h is the only successor of g, and h must start by 21, g must end by 21. So the completion time of g becomes 19,21 and the start time becomes 17,19. The next subtask to be processed will be d. It has successors f and g. Subtask f has to start by 18, so d's completion becomes 17,18, and its start becomes 8,9. The next subtask to be processed will be e. Subtask e has g and i as successors. Subtask g's latest start time is 19 and i's is 22, so subtask e becomes 10,14 for start times and 15,19 for completion times. The last subtask to be processed is a. It has successors c and d. Subtask a has to complete by 9, so the completion time will be 8,9, and its start time will be 0,1. Table 4-3 summarizes the results.

4.3 PERT—Program Evaluation and Review Technique

Table 4-3 Subtasks with Slack Time

Subtask ID	Start Time	Completion Time	Critical Path
a	0,1	8,9	
b	0	10	*
c	10	18	*
d	8,9	17,18	
e	10,14	15,19	
f	18	21	*
g	17,19	19,21	
h	21	25	*
i	21,22	24,25	

Software Pricing

- Estimates are made to discover the cost, to the developer, of producing a software system.
  - You take into account, hardware, software, travel, training and effort costs.
- There is not a simple relationship between the development cost and the price charged to the customer.
- Broader organisational, economic, political and business considerations influence the price charged.

Factors affecting Software Pricing

Factor	Description
Market opportunity	A development organization may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organization the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Cost estimate uncertainty	If an organization is unsure of its cost estimate, it may increase its price by a contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organization may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business. Cash flow is more important than profit in difficult economic times.

4.4 Software Cost Estimation

- The task of software cost estimation is to determine how many resources are needed to complete the project. Usually this estimate is in programmer-months (PM).
- There are two very different approaches to cost estimation.
  - The older approach is called LOC estimation, since it is based on initially estimating the number of lines of code that will need to be developed for the project.
  - The newer approach is based on counting function points in the project description.

ESTIMATION OF LINES OF CODE (LOC)

- The first step in LOC-based estimation is to estimate the number of lines of code in the finished project. This can be done based on experience, size of previous projects, size of a competitor's solution, or by breaking down the project into smaller pieces and then estimating the size of each of the smaller pieces. A standard approach is, for each $piece_i$, to estimate the maximum possible size, max_i, the minimum possible size, mini, and the ''best guess'' size, $best_i$. The estimate for the whole project is 1/6 of the sum of the maximums, the minimums, and 4 times the best guess:

$$\text{Standard deviation of S} = (sd^2 + sd^2 + \ldots + sd^2)^{1/2}$$
$$\text{Standard deviation of E(i)} = (\max - \min)/6$$

ESTIMATION OF LINES OF CODE (LOC)

EXAMPLE 4.8

Team WRT had identified seven subpieces to their project. These are shown in Table 4-4 with their estimates of the size of each subpiece.

Table 4-4 Subpiece Size Estimate (in LOC)

Part	Max Size	Best Guess	Min Size
1	20	30	50
2	10	15	25
3	25	30	45
4	30	35	40
5	15	20	25
6	10	12	14
7	20	22	25

ESTIMATION OF LINES OF CODE (LOC)

The estimates for each section are as follows:

$$p1(20 + 4 * 30 + 50)/6 = 190/6 = 31.6$$
$$p2(10 + 4 * 15 + 25)/6 = 95/6 = 15.8$$
$$p3(25 + 4 * 30 + 45)/6 = 190/6 = 31.6$$
$$p4(30 + 4 * 35 + 40)/6 = 220/6 = 36.7$$
$$p5(15 + 4 * 20 + 25)/6 = 120/6 = 20$$
$$p6(10 + 4 * 12 + 14)/6 = 72/6 = 12$$
$$p7(20 + 4 * 22 + 25)/6 = 133/6 = 22.17$$

The estimate for the whole project is the sum of the estimates for each section:

$$\text{Whole} = 31.6 + 15.8 + 31.6 + 36.7 + 20 + 12 + 22.17 = 170.07 \text{ LOC}$$

The estimate for the standard deviation of the estimate is as follows:

$$\text{Standard deviation} = ((50 - 20)^2 + (25 - 10)^2 + (45 - 25)^2$$
$$+ (40 - 30)^2 + (25 - 15)^2 + (14 - 10)^2 + (25 - 20)^2)^{.5}$$
$$= (900 + 225 + 400 + 100 + 100 + 16 + 25)^{.5}$$
$$= 1766^{.5} = 42.03$$

LOC-BASED COST ESTIMATION

The basic LOC approach is a formula that matches the historical data. The basic formula has three parameters:

$$\text{Cost} = \alpha * \text{KLOC}^{\beta} + \gamma$$

Alpha, α, is the marginal cost per KLOC (thousand lines of code). This is the added cost for an additional thousand lines of code. The parameter beta, β, is an exponent that reflects the nonlinearity of the relationship. A value of beta greater than 1 means that the cost per KLOC increases as the size of the project increases. This is a diseconomy of scale. A value of beta less than 1 reflects an economy of scale. Some studies have found betas greater than 1, and other studies have betas less than 1. The parameter gamma, γ, reflects the fixed cost of doing any project. Studies have found both positive gammas and zero gammas.

LOC-BASED COST ESTIMATION

EXAMPLE 4.9

Company LMN has recorded the following data from previous projects. Estimate what the parameters for the cost estimation formula should be and how much effort a new project of 30 KLOC should take (see Table 4-5).

Table 4-5 Historical Data

Proj. ID	Size (KLOC)	Effort (PM)
1	50	120
2	80	192
3	40	96
4	10	24
5	20	48

Analyzing or plotting this data would show a linear relationship between size and effort. The slope of the line is 2.4. This would be alpha, α, in the LOC-based cost estimation formula. Since the line is straight (linear relationship), the beta, β, is 1. The gamma, γ, value would be zero.

CONSTRUCTIVE COST MODEL (COCOMO)

- COCOMO is the classic LOC cost-estimation formula. It was created by Barry Boehm in the 1970s. He used thousand delivered source instructions (KDSI) as his unit of size. KLOC is equivalent. His unit of effort is the programmer-month (PM).
- Boehm divided the historical project data into three types of projects:
  1. Application (separate, organic, e.g., data processing, scientific)
  2. Utility programs (semidetached, e.g., compilers, linkers, analyzers)
  3. System programs (embedded)
- He determined the values of the parameters for the cost model for determining effort:
  1. Application programs: $\mathrm{PM} = 2.4 * (\mathrm{KDSI})^{1.05}$
  2. Utility programs: $\mathrm{PM} = 3.0 * (\mathrm{KDSI})^{1.12}$
  3. Systems programs: $\mathrm{PM} = 3.6 * (\mathrm{KDSI})^{1.20}$

CONSTRUCTIVE COST MODEL (COCOMO)

EXAMPLE 4.10

Calculate the programmer effort for projects from 5 to 50 KDSI (see Table 4-6)

Table 4-6 COCOMO Effort

Size	Appl	Util	Sys
5K	13.0	18.2	24.8
10K	26.9	39.5	57.1
15K	41.2	62.2	92.8
20K	55.8	86.0	131.1
25K	70.5	110.4	171.3
30K	85.3	135.3	213.2
35K	100.3	160.8	256.6
40K	115.4	186.8	301.1
45K	130.6	213.2	346.9
50K	145.9	239.9	393.6

CONSTRUCTIVE COST MODEL (COCOMO)

- Boehm also determined that in his project data, there was a standard development time based on the type of project and the size of the project.
- The following are the formulas for development time (TDEV) in programmer-months:

1. Application programs: $\text{TDEV} = 2.5 * (\text{PM})^{0.38}$
2. Utility programs: $\text{TDEV} = 2.5 * (\text{PM})^{0.35}$
3. Systems programs: $\text{TDEV} = 2.5 * (\text{PM})^{0.32}$

CONSTRUCTIVE COST MODEL (COCOMO)

EXAMPLE 4.11

Calculate the standard TDEV using the COCOMO formulas for projects from 5 to 50 KDSI (see Table 4-7).

Table 4-7 COCOMO Development Time

Size	Appl	Util	Sys
5K	6.63	6.90	6.99
10K	8.74	9.06	9.12
15K	10.27	10.62	10.66
20K	11.52	11.88	11.90
25K	12.60	12.97	12.96
30K	13.55	13.93	13.91
35K	14.40	14.80	14.75
40K	15.19	15.59	15.53
45K	15.92	16.33	16.25
50K	16.61	17.02	16.92

FUNCTION POINT ANALYSIS

- The idea of function points is to identify and quantify the functionality required for the project. The idea is to count things in the external behavior that will require processing. The classic items to count are as follows:
  - Inputs
  - Outputs
  - Inquiries
  - Internal files
  - External interfaces
- *External Inquiries(EQ)* are request-response pairs that do not change the internal data. For example, a request for the address of a specified employee is an inquiry. The whole sequence of asking, supplying the name, and getting the address would count as one inquiry.
- *External Inputs(EI)* are items of application data that is supplied to the program. The logical input is usually considered one item and individual fields are not usually counted separately. For example, the input of personal data for an employee might be considered one input.

FUNCTION POINT ANALYSIS

- *External Outputs(EO)* are displays of application data. This could be a report, a screen display, or an error message. Again, individual fields are usually not considered separate outputs. If the report has multiple lines, for instance, a line for each employee in the department, these lines would all be counted as one output. However, some authorities would count summary lines as separate outputs.
- *Internal Logical Files(ILF)* are the logical files that the customer understands must be maintained by the system. If an actual file contained 1000 entries of personnel data, it would probably be counted as one file. However, if the file contained personnel data, department summary data, and other department data, it would probably be counted as three separate files for the purposes of counting function points.
- *External Interface Files(EIF)* are data that is shared with other programs. For example, the personnel file might be used by human resources for promotion and for payroll. Thus, it would be considered an interface in both systems.

Counting Unadjusted Function Points

- The individual function point items are identified and then classified as simple, average, or complex. The weights from Table 4-8 are then assigned to each item and the total is summed. This total is called the unadjusted function points. There is no standard for counting function points.
- For example, one difference between approaches to counting function points is related to summary lines at the bottom of reports. Some software engineers feel that a summary line means another output should be counted, while others would only count the main items in the report.
- Working together and reviewing other function point analyses will help build that consistency.
- Note: Try to make your function points consistent with effort necessary for processing each item.

Table 4-8 Function Point Weights

	Simple	Average	Complex
Outputs	4	5	7
Inquiries	3	4	6
Inputs	3	4	6
Files	7	10	15
Interfaces	5	7	10

Counting Unadjusted Function Points

EXAMPLE 4.12

The department wants a program that assigns times and rooms for each section and creates a line schedule for the courses. The department has a list of sections with the name of the assigned professor and the anticipated size. The department also has a list of rooms with the maximum number of students each room will hold There are also sets of classes that cannot be taught at the same time. Additionally, professors cannot teach two courses at the same time.

This program is much more difficult than the complexity of the inputs and outputs. It has two main inputs: the file with the list of sections, assigned professor, and anticipated size, and the file with the list of rooms with the maximum size. These two files, although simple to read, will be difficult to process, so they will be rated complex. There will be an additional file with the sets of classes that cannot be taught at the same time. Again, this file is simple in structure but will be difficult to process. The last line has a restriction that is not an input or output.

There is an output, the line schedule. This is a complex output. There are no inquiries or interfaces mentioned, nor any mention about files being maintained.

Counting Unadjusted Function Points

- Procedure to count UFP(Unadjusted Function Points)
  - Step 1: The individual function point items are identified and counted by EI, EO, EQ, ILF, EIF.
  - Step 2: The individual function point items are classified as simple, average, or complex.
  - Step 3: Weighted point for each function point item is calculated by multiplying its counted point and its weight.
  - Step 4: UFP is calculated by summing the weighted points.

PRODUCTIVITY

- One of the important measures is the productivity of the software developers. This is determined by dividing the total size of the finished product by the total effort of all the programmers. This has units of LOC/programmer-day. An alternative is to measure the productivity in terms of function points per programmer-day.
- Note that productivity includes all the effort spent in all phases of the software life cycle.

PRODUCTIVITY

EXAMPLE 4.13

Company XYZ spent the following effort for each life cycle phase of the latest project (see Table 4-9). Calculate the effort in terms of LOC/programmer-day and in terms of function points/programmer day. The function point estimate was 50 unadjusted function points. The finished project included 950 lines of code.

Table 4-9 Effort During Phases

Phase	Programmer-Days
Requirements	20
Design	10
Implementation	10
Testing	15
Documentation	10

The total effort was 65 programmer-days.

This gives a productivity of 950/65 = 14.6 lines of code/programmer-days.

Using unadjusted function points (fp), the productivity is 50 fp/65 days = 0.77 fp/programmer-days.

EVALUATING ESTIMATIONS

- To evaluate estimations, a measure needs to be calculated. Tom DeMarco proposed the *estimate quality factor* (EQF). DeMarco defines the EQF as the area under the actual curve divided by area between the estimate and the actual value. This is the inverse of the percentage error or the mean relative error. Thus, the higher the EQF, the better was the series of estimates. DeMarco said that values over 8 are reasonable.

EXAMPLE 4.11

The following estimates were given for a project that cost 3.5 million dollars when it was completed after 11.5 months:

Initial	1.5 months	5.5 months	8 months
2.3 million	3.1 million	3.9 million	3.4 million

The total area is 11.5 months times 3.5 million = 40.25 million month-dollars. The difference between the actual curve and the estimate is $|2.3 - 3.5| * 1.5 + |3.1 - 3.5| * 4 + |3.9 - 3.5| * 2.5 + |3.4 - 3.5| * 3.5 = 4.75$ million month-dollars. The ratio is $40.25/4.75 = 8.7$.