Chapter 5
Planning Activities and Predicting Costs

In Chapter 3 we saw that development processes can provide a broad work-flow to govern the development of a software product. In Chapter 4 we covered the various conventions and technologies that can help developers to successfully collaborate on source code. Even with those tools, there still needs to be an understanding amongst the developers of who will do what, and when. In a project with a tight time and resource constraints, how much time can be allocated to particular activities? Which activities are the best ones to prioritise? At the end of the day, how much is the whole endeavour going to cost, and how much time will it require?

Such questions are crucial to quality assurance. If the time and effort required for a project is not estimated properly, it can lead to huge cost overruns. A 2011 study of 1,1471 IT projects [52] established that 27% of IT projects are subject to cost-overruns. More worryingly, one in six projects was subject to cost-overruns of more than 200%, and schedule overruns of 70%. In financial terms, such overruns can be disastrous.

Although certain aspects of development processes can help to guard against such overruns (c.f. the use of time-boxed iterations by IID and SCRUM), these can only succeed if the developers are able to plan their work and predict the resources required. Over the years several approaches have emerged to support these activities.

This chapter presents these approaches in two parts. Section 5.1 presents approaches that support planning at a lower-level; determining when what activities should be undertaken. For this we present two general project-management techniques: PERT and Gantt charts. In section 5.2 we present techniques that are specifically geared towards predicting the cost of a project (or a portion thereof). These techniques are not concerned with the low-level sequencing of activities, but are more statistical in nature; examining previous experience and cost data to make a valid prediction of the overall cost of a project.

N. Walkinshaw, *Software Quality Assurance*, Undergraduate
Topics in Computer Science, DOI 10.1007/978-3-319-64822-4_5

5.1 Planning

The cost and time required to develop a project are often intricately interlinked. Producing a high quality software system in a short amount of time can impose huge pressures on developers, taking them away from other projects or requiring the recruitment of new staff (and thus leading to higher costs). Allowing for a longer time-frame can relieve pressures on staff, can provide more scope for testing and other quality assurance activities, leading to higher quality software at a potentially lower cost.

Ultimately, in either case, success comes down to proper planning. A plan needs to set out the (anticipated) *resources* that will be required throughout the course of a project. It needs to set out when certain tasks need to be achieved. It needs to also, set of priorities amongst tasks, and set out which tasks potentially have some scope for leeway to allow for overruns.

5.1.1 Program Evaluation and Review Technique (PERT)

At this level, a software engineering project is akin to any other conventional engineering project. Accordingly, the most popular planning techniques were not specifically developed for software projects, but can be applied in the same manner. In this section we cover one of the most popular Program Evaluation and Review Technique (PERT).

The PERT technique was developed for the U.S. Navy in 1957 to support their development of the Polaris submarine-launched nuclear weapons system [92]. Its main selling point was the ability to explicitly incorporate uncertainty as far as the scheduling of individual tasks is concerned. This makes it especially appealing from a software engineering point of view where, as we shall see in Chapter 5, predicting the duration it takes to develop a software module is fraught with uncertainty.

The PERT technique starts off with a table that captures all of the essential information about the project. The project is split into its essential *activities*. Each activity is associated with the following attributes:

- **Predecessor:** Any other activities that must be completed in order for this activity to start.
- **Time estimates:**
  - **Optimistic** (*o*)**:** The duration that this activity will take in an ideal setting where there are no hitches and the developers are able to make good progress.
  - **Normal** (*n*)**:** The duration that this activity will take under 'normal' circumstances.
  - **Pessimistic** (*p*)**:** The duration that this activity will take if problems are encountered along the way.

- **Expected time:** This is derived from the above time estimates as $\frac{o+4n+p}{6}$. In other words, it takes the average of o, n, and p, but gives n a weighting that is four times greater than the optimistic and pessimistic estimates[1].

The time-units used depend on the broader context of the project. For smaller projects with a more granular plan the unit could be person-hours. For larger projects it might be days or even weeks of developer-time. For our examples we assume days.

		Time Estimates			
Activity	**Predecessor**	**Optimistic**	**Normal**	**Pessimistic**	**Expected Time**
A: Develop storage format	D	2	3	4	3
B: Develop file reader	A	3	5	6	4.83
C: Develop file writer	A	3	4	7	4.33
D: Develop core data structure	-	3	5	10	5.5

Table 5.1 Example table of activities and time estimates, for a small component to load and save data.

An example table of activities is shown in Table 5.1. These activities are concerned with the activities that are involved in the development of a module for reading and writing data to the disk. There are some pertinent dependencies between the activities. The components for writing-to and reading from the file on disk (activities B and C) cannot be achieved until we know what the data format is that we are dealing with (which is developed in activity A). However, this cannot be achieved until we know what the underlying data structure(s) in the program are – which data elements will it be necessary for the program to be able to store and access in a persistent manner (activity D).

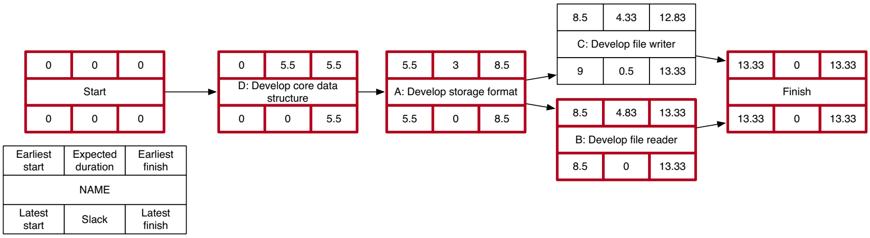

Fig. 5.1 PERT chart for activities in Table 5.1.

Once the activities have been entered into the table, they can be displayed as a PERT chart. This is created in three phases. Firstly, the 'network diagram' is created to highlight the sequential dependencies. This is achieved by creating a 'start' node, one node for each of the activities, and a finish node. The nodes are then connected

[1] It is not clear that this formula is clearly justified; i.e. *why* the weighting of four is chosen for the normal estimate.

according to the ‘predecessor’ activities set out in the table. The Start node is connected to the activity with no predecessors, and then the dependencies are traced out. Any activities with no successors are connected to the ‘finish’ node.

The next step is to identify the time-constraints that govern the activities. To facilitate this, we use the following structured labels to annotate the activities.

- Earliest start time and Latest start time
- Expected duration (as calculated in from the time estimates) and slack-time
- Earliest finish time and latest finish time

We populate the PERT chart with these values as follows:

1. In the ‘start’ node, set all of the values to zero.
2. For each of the nodes that follow on from the start node, set the earliest start time to zero, and set the earliest finish time to the expected duration of that activity.
3. Trace out the subsequent dependent activities, setting their earliest start time to the earliest finish time of the preceding activities, and calculating the earliest finish times by adding the expected duration to the earliest start time.

    - If you have the situation where you have a node with multiple preceding activities (such as the Finish node in our case) the earliest start time is the maximum of the “early finish” times of any of the preceding nodes.

4. In the ‘Finish’ node, set the latest finish time and latest start times to the earliest finish time.
5. Trace back to the preceding activities, and set each of their latest finish times to the latest start time of the finish node. Calculate their latest start time by subtracting the expected duration from the latest finish time.

    - In situations where several nodes are preceded by a single node, the latest finishing time of that node is set to the minimum latest start time of the succeeding nodes.

6. Calculate the slack for all of the nodes by subtracting the earliest finishing time from the latest finishing time.

Finally, we are able to calculate the Critical Path . This constitutes the path through the PERT chart where there is no slack. Where any delay to an activity will have immediate knock-on effects for the rest of the project and will result in a delay to the overall finishing time. On our PERT chart this is highlighted in red.

The critical path is particularly valuable when it comes to prioritisation. From a management perspective, if two activities are being worked on concurrently, where one is on the critical path and the other has some slack-time, it makes sense to prioritise the critical project. For example, by moving relevant staff from the from the less critical activity to the critical one.

5.1.2 Gantt Charts

PERT charts are a useful means by which to elicit the key dependencies and time constraints that will underpin a project. However, the PERT chart can be difficult to interpret. The chart does not visually convey the explicit *flow* or *duration* of the activities. One visual tool that can be used to better convey the timings of the activities is the Gantt chart. The Gantt chart is named after Henry Gantt, who devised them in the early 1900s (their specific origins are dates are not known [134]). They came to prominence through Gantt's work during the First World War, and became widespread project planning tools in the mid 1920s.

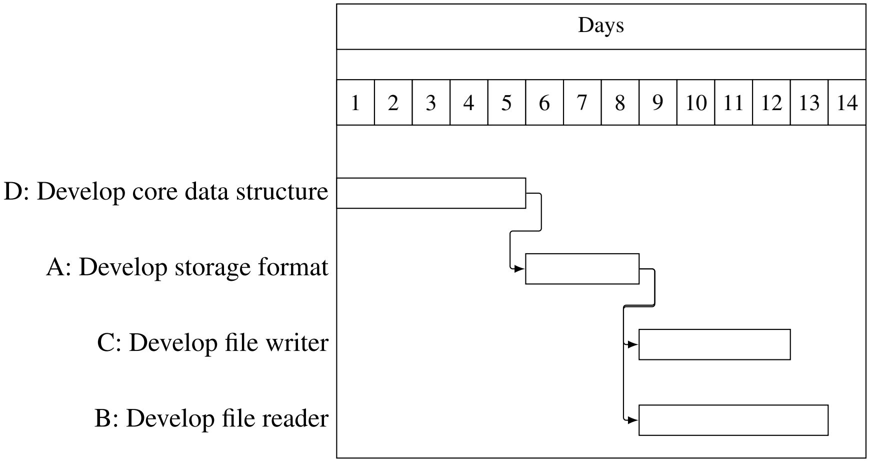

Fig. 5.2 Gantt chart corresponding to the activities in Table 5.1.

Figure 5.2 shows an example of a Gantt chart corresponding to the activities in Table 5.1 (using the expected durations). For the sake of illustration, decimal places in the durations are rounded to the nearest full day. The chart sets out the flow of activity (what happens when), and how long each activity is going to take. Gantt charts often include two different types of arrow: One type to represent the sequential flow between activities, and another to represent the dependency of one activity on another (in our example the flow and dependencies overlap anyway). It is trivially also possible to highlight the critical path on a Gantt chart (by simply highlighting the relevant activities on the path).

Even though they both represent the same information, Gantt and PERT charts are nevertheless complementary in nature. Whereas Gantt charts are purely visual, PERT charts also provide a basis upon which to calculate some of the data that might go into a Gantt chart; the durations, and essential dependencies (which in turn inform the possible flow of activities).

5.2 Predicting Costs

Effort[2] prediction is crucial when it comes to planning how a software system is going to be developed. It is vital to determine the resources that are required, and how this matches up ot the resources that are available. If the software is being paid for, it is necessary to agree upon a price for the development, and this can only be done in a reasonably reliable way if one can draw upon some prior experience.

The nature and fidelity of effort estimation depends on the type and amount of prior data that is available. At the simplest end of the spectrum, one might only have an idea of how large and complex the system ought to be. On the other hand, one might have both such an estimate, along side a wealth of cost data from previous projects.

5.2.1 Base Models

The simplest approach is to predict the cost in two steps. First of all, one derives an estimate of software complexity or size (S), by deriving a measure such as Albrecht's Function Points or an overall estimate of LOC from the requirements. The act of estimation is then reduced to estimating the cost of developing a single unit a (e.g. cost per LOC or cost per function point). So the cost E is computed as:

$$E = a * s \tag{5.1}$$

For example, if you estimate that your project will amount to 1,230 LOC, and you estimate the cost per LOC to be £25, then the total cost $E = 25 * 1250 = £30,750$.

Often, the cost of a model does not merely depend on "size". There can be a base-cost to a project before a single line of code has been written - with the preliminary requirements elicitation, the initial architecture, design, recruitment, etc. To account for this, we can add an additional parameter to the model, c, so that the estimate becomes:

$$E = a * s + c \tag{5.2}$$

In this case, if one were to plot the cost estimate as a line, c would be the intercept – the point at which the line crosses the y axis.

In reality, the relationship between size and effort is not linear. After all, every line of code (let alone entire function) that is added to a project presents the developer with an additional responsibility to maintain and test as well. To address this, an additional 'scale' coefficient b is added, resulting in:

$$E = a * s^b + c$$

[2] In this context, the terms "cost" and "effort" are used synonymously.

In this case $b > 1$ indicates a non-linear increase in cost per unit (i.e. per LOC). Conversely, $b < 1$ indicates a decrease, though this is highly unlikely to be the case in a typical project.

5.2.2 Parameter Fitting by Linear Regression

But where do the values for parameters such as a, b, and c come from? Unless you have some prior experience, it has to come out of pure intuition, and this is unlikely to be particularly reliable. However, if you do have experience – if you have recorded previous project costs and sizes, it becomes possible to make a more educated guess. This is the setting that broadly sets the scene for most of the techniques in software cost estimation.

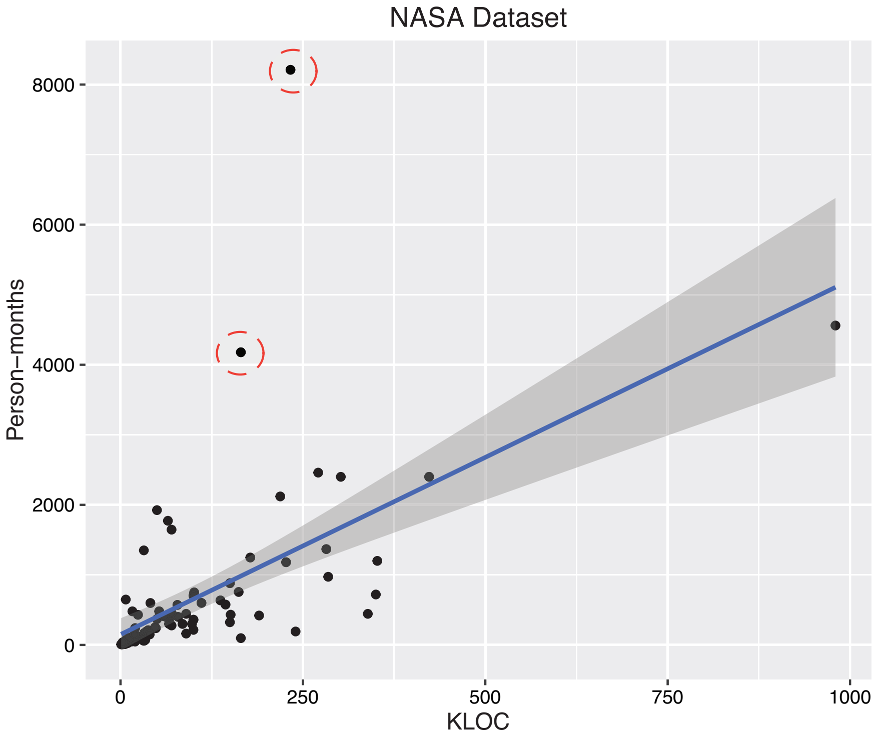

Fig. 5.3 Linear Regression on cost-effort data from the NASA'93 dataset. $a = 5.059$ and $c = 148.8$

The approach is best illustrated visually, and by example. To illustrate this we use a dataset collated by NASA in 1993, which maps the amount of effort in person-months to KLOC (amongst other things) for 101 data-points, spanning eight projects[3]. The scatter-plot for this data is shown in Figure 5.3. Essentially cost estimation amounts to identifying a “model” that fits this data, and which can thus be used to predict the effort for projects of other sizes.

The task now is to identify a “model” that can explain this data, and that we can therefore use to predict the cost of our project. In the simplest case, we can assume that the model in question is a straightforward line (in this case we assume that $b = 1$).

The approach of finding the necessary values for a and c to best fit a set of data is known as *Linear Regression*. We will not cover the specific mechanics of Linear Regression in this book - there are plenty of references and tools that will do this for you. In our case, if we apply Linear Regression to the data in Figure 5.3, we end up the line that is plotted in the figure – which has the coefficients $a = 5.059$ and $c = 148.8$.

Exercise: *The model fits a lot of the data points quite well. However, it fares less well with some of the “outliers” highlighted in the plot. Why might the data points not fit a straight line? What would the implications have been, had NASA adopted this linear model?*

As shown above, the model of a simple straight line for modelling cost or effort can be somewhat coarse. For example, one simplifying assumption made by the straight-line cost model is that $b = 1$ – that the cost is directly relative to a fixed cost-per Line of Code (or whatever the unit of software size). In reality, it is likely that $b > 1$; that for every additional line of code there is an additional cost to bear, in terms of efforts involved in activities such as maintenance, testing, documentation, etc.

5.2.3 COCOMO

The linear regression can fit a straight line to existing data, and provides an rough basis upon which to begin approximating the costs of a project. However, a straight line does not allow us to account for other ‘non-linear’ factors, such as the scale factor b. Also, fitting the parameters by approaches such as Linear Regression can be tricky because one might not have access to the sort of historical data that is required to yield a useful prediction.

To address this, Boehm developed the Constructive Cost Model (COCOMO) effort prediction framework in 1981 [23]. Instead of requiring historical data, the model provides certain parameters that, collectively, should characterise the development task at hand. It comes in three levels:

[3] You can download this data yourself from the PROMISE software repository [6]

1. **Basic:** The project is in its very early stages and has been subject only to some very basic requirements capture.
2. **Intermediate:** The requirements have been collected to a reasonable degree of detail.
3. **Detailed:** The requirements have been collected and the system has been designed.

For each level, COCOMO provides additional parameters that can be populated by the user to provide a (hopefully) increasingly accurate prediction of the effort required. Here we present the Basic and Intermediate stages[4].

Over the years, COCOMO (and its successor COCOMO II) have become widely used within industry, especially in the Aerospace domain by organisations such as NASA.

5.2.3.1 Basic Model

The basic model for COCOMO provides three models, to predict three aspects of the effort that a project is going to require. The effort E is computed by the familiar equation we have covered previously:

$$E = a * s^b \tag{5.3}$$

This effort value contributes to the calculation of the development time, which is computed as follows:

$$D = c * E^d \tag{5.4}$$

Both of these values can then be used to calculate the number of people required:

$$P = E/D \tag{5.5}$$

To use these equations, the user has to provide s (the estimated size of the final system in KLOC), and has to select the values of four coefficients (a, b, c, and d). The COCOMO model provides suggested values for these parameters. These values were derived from data collected from 63 software development projects in the 70s[5]. These projects were largely from a single organisation, along with a selection of others from university courses and other organisations.

The selection of a suitable set of variables first of all depends on the "type" of project. This is decided as one of the following:

1. **Organic:** Small teams of experienced developers with flexible requirements.

[4] The detailed stage is a very Waterfall-model specific adaptation that refines the prediction in a way that is sensitive to the various development stages.

[5] The original COCOMO dataset can be downloaded here: `http://openscience.us/repo/effort/cocomo/coc81.html`

2. **Semi-detached:** Medium teams, mixed experience, working towards a mixture of tight and flexible requirements.
3. **Embedded:** The software is developed towards tight constraints.

Based on this categorisation, the values for a, b, c and d are shown in Table 5.2.

Development context	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.2	2.5	0.32

Table 5.2 COCOMO coefficients for the ”Basic” model.

5.2.3.2 Intermediate COCOMO

The intermediate version of COCOMO incorporates some additional factors that can have a bearing on the time and effort required within a project, but which were not considered in the basic version. These “cost drivers” are shown in Figure 5.4.

- Product attributes
  - Required software reliability
  - Size of application database
  - Complexity of the product
- Hardware attributes
  - Run-time performance constraints
  - Memory constraints
  - Volatility of the virtual machine environment
  - Required turnabout time
- Personnel attributes
  - Analyst capability
  - Software engineering capability
  - Applications experience
  - Virtual machine experience
  - Programming language experience
- Project attributes
  - Use of software tools
  - Application of software engineering methods
  - Required development schedule

Fig. 5.4 Cost drivers for Intermediate COCOMO

For each of these cost drivers, the developer is required to rate them on an ordinal scale from 1 to 6, where 1 is “very low” and 6 is “extra high”. Each combination of factor and rating is given a weighting, as shown in Table 5.3. The “Effort Adjustment Factor” (EAF) is calculated as the product of the weightings attributed to all of the cost drivers.

The Intermediate COCOMO uses this EAF to produce a more refined effort estimation:

Category	Cost drivers	Very low	Low	Nominal	High	Very High	Extra High
Product	Req. Reliability	0.75	0.88	1.0	1.15	1.4	
	Database size		0.94	1.0	0.08	1.16	
	Complexity	0.70	0.85	1.0	1.15	1.3	1.65
Hardware	RT Perf.			1.0	1.11	1.3	1.66
	Memory			1.0	1.06	1.21	1.56
	Volatility of VM		0.87	1.0	1.15	1.3	
	Turnabout time		0.87	1.0	1.07	1.15	
Personnel Attributes	Analyst	1.46	1.19	1.0	0.86	0.71	
	Apps.	1.29	1.13	1.0	0.91	0.82	
	Soft. Eng.	1.42	1.17	1.0	0.86	0.70	
	Virt. Machine	1.21	1.1	1.0	0.9		
	Prog. Lang.	1.14	1.07	1.0	0.95		
Project	SE Methods	1.24	1.1	1.0	0.91	0.82	
	Tools	1.24	1.1	1.0	0.91	1.83	
	Schedule	1.23	1.08	1.0	1.04	1.1	

Table 5.3 Weightings for Intermediate COCOMO “cost drivers”.

$$E = a * s^b * EAF \tag{5.6}$$

The other calculations (for development time and personnel) are calculated as in the basic model. The coefficients for a, b, c and d are chosen from the same set of values (as shown in Table 5.2).

Exercise: *Compare and contrast Intermediate COCOMO with Albrecht’s Function Point model. Can you identify any common weaknesses between them?*

5.2.3.3 COCOMO II

The application of COCOMO is problematic. Looking at the entire COCOMO model, it provides a set of fixed parameters for a, b, c, d, as well as the 15 cost drivers. It has derived values for these 19 parameters from a relatively small selection of projects (only 63). Furthermore, the projects in question that were used to derive these values were of a relatively specific nature; they were all from the 70s; most likely written in languages such as C and Pascal, with development methodologies that have since become largely outdated, such as the Waterfall model. Applying these values in a modern context, to an agile project that is being developed with modern tooling, by developers with different skill sets, adopting “new” language paradigms such as Object-Oriented development, risks producing results that are misleading.

To address this weakness, Boehm collated a new, larger set of data (amounting to 161 projects). He used this to fit variables for a new COCOMO II model [20], which is designed to be applied to software systems that are developed in a more modern context. Their intention is to take into account factors such as reuse, automated programming, the integration of COTS components, etc., which were not explicitly accounted for by the original COCOMO model.

The equation underpinning the COCOMO II model is as follows:

$$E = a * \prod_i EM_i * s^{b+0.01*\Sigma_j SF_j} \tag{5.7}$$

Here, a, s, and b are the same as the variables that were used in the original COCOMO.

The SF represents a set of 'scale-factor' values that are added up to produce an overall scale factor. These are obtained by rating a set of factors on a scale from 1-6. These ratings are chosen according to Table 5.4.

Definition ***(abbreviation)***	**Low (1,2)**	**Medium (3,4)**	**High (5,6)**
Development flexibility *(flex)*	Process rigorously defined	Some guidelines, can be relaxed	Only general goals
Process Maturity *(pmat)*	CMM Level 1	CMM Level 3	CMM Level 5
Precedentedness *(prec)*	Never build this kind of software before	Quite new	Thoroughly familiar
Architecture or risk resolution *(resl)*	Few interfaces defined or few risks eliminated	Most interfaces defined, most risks eliminated	All interfaces defined and all risks eliminated
Team cohesion *(team)*	Very difficult interactions	Basically cooperative	Seamless interactions

Table 5.4 Scale factors for COCOMO II

EM is a set of "effort multipliers". These are chosen on a similar basis to the scale factors, and are shown in Table 5.5. The actual coefficient values that are associated with each of these ratings for both SF and EM are shown in Table 5.6.

Once the EM and SF elements have been provided, the effort E can be calculated, which can feed into the calculation of developer time and number of people required as in equations 5.4 and 5.5. Boehm recognised that the given coefficients may not be appropriate to an arbitrary setting. It is unlikely that the data, even in its expanded form of 161 projects, would not adequately apply to every applied setting. Accordingly, it is customary to tune COCOMO. This can be achieved if an organisation has just a few examples, and is commonly accomplished by varying variables a and b whilst holding the other values constant.

Definition	**Low (1,2)**	**Medium (3,4)**	**High (5,6)**
Analyst capacity *(acap)*	Worst 35%	35% – 90%	best 10%
Applications experience *(aexp)*	2 months	1 year	6 years
Product complexity *(cplx)*	E.g. Simple read / write statements	E.g. Use of simple interface widgets	E.g. Performance critical embedded system
Database size (DB bytes / SLOC) *(data)*	10	100	1000
Documentation *(docu)*	Many life-cycle phases not documented		Extensive reporting for each life-cycle phase
Language and tool experience *(ltex)*	2 months	1 year	6 years
Programmer Capability *(pcap)*	Wost 15%	55%	best 10%
Personnel Continuity (% turnover per year.) *(pcon)*	48%	12%	3%
Platform experience *(plex)*	2 months	1 year	6 years
Platform volatility *(pvol)*	Major change every 12 months, minor change every month.	Major change every 6 months, minor change every 2 weeks.	Major change every 2 weeks, minor change 2 days.
Required reliability *(rely)*	Errors a slight inconvenience	Errors are easily recoverable	Errors can risk human life
Required reuse *(ruse)*	none	Multiple program	Multiple product lines
Dictated program schedule *(sced)*	Deadlines moved to 75% of original estimate	No change	Deadlines moved back to 160% of original estimate
Multi-site development *(site)*	Some - contact by phone and mail	some email	Interactive multimedia
Required % of available RAM *(stor)*	N/A	50%	95%
Required % of CPU *(time)*	N/A	50%	95%
Use of software tools *(tool)*	edit, encoding, debug		Integrated with life-cycle

Table 5.5 Effort Multipliers for COCOMO II

Category		Vey Low	Low	Norm	High	Very High	Extra High
Scale Factors	Prec	6.20	4.96	3.72	2.48	1.24	0.00
	Flex	5.07	4.05	3.04	2.03	1.01	0.00
	Resl	7.07	5.65	4.24	2.83	1.41	0.00
	Team	5.48	4.38	3.29	2.19	1.10	0.00
	Pmat	7.80	6.24	4.68	3.12	1.56	0.00
Effort Multipliers	Rely	0.82	0.92	1.00	1.10	1.26	–
	Data	–	0.90	1.00	1.14	1.28	–
	Cplx	0.73	0.87	1.00	1.17	1.34	1.74
	Ruse	–	0.95	1.00	1.07	1.15	1.24
	Docu	0.81	0.91	1.00	1.11	1.23	–
	Time	–	–	1.00	1.11	1.29	1.63
	Stor	–	–	1.00	1.05	1.17	1.46
	Pvol	–	0.87	1.00	1.15	1.30	–
	Pcon	1.29	1.12	1.00	0.90	0.81	–
	Acap	1.42	1.19	1.00	0.85	0.71	–
	Pcap	1.34	1.15	1.00	0.88	0.76	–
	Apex	1.22	1.10	1.00	0.88	0.81	–
	Plex	1.19	1.09	1.00	0.91	0.85	–
	Ltex	1.20	1.09	1.00	0.91	0.84	–
	Tool	1.17	1.09	1.00	0.90	0.78	–
	Site	1.22	1.09	1.00	0.93	0.86	0.80
	Sced	1.43	1.14	1.00	1.00	1.00	–

Table 5.6 COCOMO II Coefficients

5.2.4 Planning Poker

So far, all of the cost-prediction approaches considered have been ‘model-based’. We are assuming that there is a hidden relationship at play, between certain properties of the software system such as its size and complexity, and the ultimate cost. However, not all cost-prediction approaches are based on this framework.

Planning poker is one cost-prediction approach that is *not* model-based. As the name suggests, Planning poker is based on a ‘game’ that several developers play amongst themselves. The approach is especially popular in agile-development contexts, and is an informal variant of the Wideband-Delphi approach proposed by Boehm in 1981 [23].

Planning poker provides a structured protocol that aims to elicit a consensus from the developers as to how much effort a piece of work is going to take. Each developer is given a set of numbered cards, where the numbers fit some non-linear scale[6], along with a wild-card “?”. For a given user story, each developer picks a card that corresponds to their estimation of how much effort a story will take. Units here are commonly referred to as ‘story points’. These are intended to measure the complexity or size of the task (but are not tied to a unit of time).

To play the game, each player places the card representing their estimate face-down on the table (to prevent biasing other players). Once the cards have been turned

[6] The scale often follows the Fibonacci sequence: 1, 3, 5, 8, 13, 20, ...

over, the players with the highest and lowest values are given the option of arguing why they believe they are right. Once they have made their cases, the entire process is repeated; players place their cards afresh until a consensus is reached.

There have been few studies into the accuracy of planning-poker, but those that have been carried out have been supportive. A study in 2008 by Moløkken *et al.* [98] on an industrial software development project indicates that planning poker leads to group-estimates that are more conservative than simply taking the averages of individual estimates, and that these consensus estimates were more accurate than individual estimates.

5.2.5 Uncertainty and Predictive Accuracy

One crucial factor in predicting software cost is the amount of reliable information that can be used about the software project in question. This amount of information increases as software development progresses. At the earliest stage, before requirements have been properly elicited, any estimates are bound to be less certain than they are in later stages.

This is one of the key sources of error in the use of, for example, the linear regression models above. In order to produce a cost estimate, it is still necessary to produce an estimate of the size of the final system. Without a proper grasp on the requirements, such a guess (and any cost estimate derived from it) has to be treated with extreme caution.

This problem was characterised by Barry Boehm as the “Cone of Uncertainty”, shown in Figure 5.5. He suggests that predictions that are made in the very initial stages can be a quarter of the actual cost if overly optimistic, or four times the actual cost if overly pessimistic.

We have seen that cost estimation is one of the most important factors in quality assurance. Without the ability to anticipate costs properly, a project is doomed. Developers are not given the time and resource necessary to fulfil their obligations, the project becomes subject to overbearing time and cost pressures that, ultimately, undermine quality, and can even lead to projects being abandoned.

The “cone of uncertainty” goes some way towards explaining why effort estimation is difficult. Early on in a project, little is known, and the estimate is necessarily approximate. This can only become more reliable as the blanks are filled in once the project has had a chance to mature.

It also gives rise to a more fundamental question: How accurate is COCOMO? How does it compare to more recent proposed approaches, or more basic approaches such as the simple Lines of Code function in Equation 5.1?

These questions, and others, were posed by Menzies *et al.* (including COCOMO’s originator Barry Boehm) in a recent paper [96]. To answer the above questions, they carried out a comprehensive study, comparing the performance of COCOMO and a raft of more recent techniques on a range of prediction tasks. Interestingly, their analysis showed that, despite its age and apparently biased coefficient

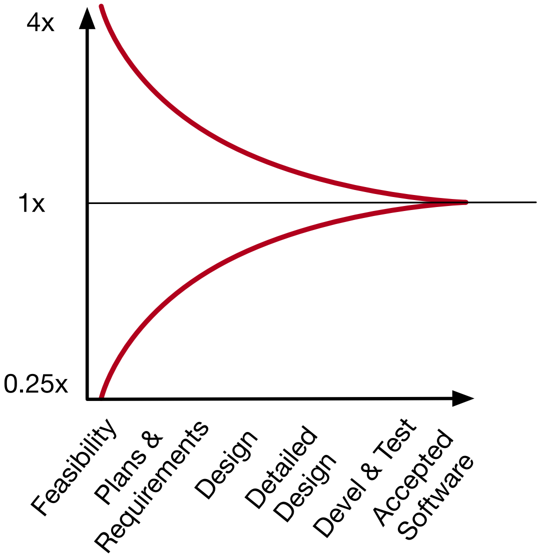

Fig. 5.5 Boehm's Cone of Uncertainty.

values, no technique outperformed COCOMO II. It was far more accurate than measures that are based upon Lines of Code alone. The key to accuracy, they found, is the choice of suitable data for calibrating the model, as opposed to the model itself.

5.2.6 Keeping Track of Progress

Once the cost of an activity has been estimated, it is important to monitor whether its actual cost reflects the prediction or not. If the prediction was wrong, it is better to realise this early on so that any miscalculations can be factored into subsequent planning iterations. For this we can refer to two useful notions that are particularly prevalent in agile software development: Team Velocity and Burndown Charts.

Team Velocity is the estimated speed at which a development team works. In an agile context it can be described as *"Number of story-points completed per sprint"*[7]. The velocity of a sprint is calculated by adding up the story point estimates for stories that were successfully completed in that sprint.

[7] We use the 'story point' unit here, but this could be replaced with whatever metric is being used to predict time or effort.

Exercise: *Having played a few rounds of Planning Poker, your team has established that the total number of story points for all of the stories amounts to 203 story points. Having completed a sprint, your team has implemented 5 stories, which amount to 25 story points. What is their velocity, and (assuming this velocity continues) how long will they take to complete the software?*

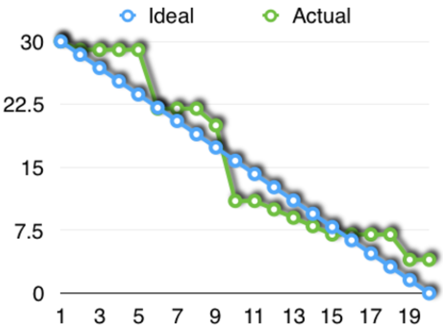

Fig. 5.6 Example burndown chart. Here, a sprint takes 20 days, and the total estimate of effort for the sprint amounts to 30 story points.

The velocity and the amount of work left to do can be visualised with Burndown charts. An example is shown in Figure 5.6. Burndown charts are quite flexible, and there is no prescriptive way in which they are to be used. They can be used to visualise progress throughout the lifetime of a project, or just for the day-by-day progress of a sprint.

The starting point is the total amount of effort that has been estimated. This can be achieved by totting up the total number of story-points estimated (or whatever other unit of effort is used). The length of the x-axis is determined by the amount of time planned for the given iteration or sprint (we refer to this number as x here), and is usually divided into days. The “ideal” line is simply a straight line from the total estimated value at day 1 to 0 effort at day x. The “actual” accomplishment is measured by the number of units that are deemed to have been completed after each day. The idea is that the burn-down chart is visible throughout the project, and is updated at regular intervals (e.g. every day).

5.3 Key Points

- **Coming in on time and on budget is a key consideration when it comes to the quality of a software system.** A huge portion of IT projects fail in this respect; a 2011 survey indicated that 27% of projects are subject to cost-overruns, and that one in six suffer cost-overruns of over 200%.
- **PERT charts were established in the late 50 so support project management activities.** They combine a simple form of cost-estimation with dependence analysis to compute dependency diagrams, where the estimated start and end-point of each activity can be predicted.
- **Gantt charts show the sequential activity within a project.** In doing so they complement PERT charts, which only show dependencies, but do not necessarily visualise the duration and actual order of activities.
- **There are several approaches to predicting the cost of a project. In most cases, techniques try to infer the cost of a project from the costs of previous projects.** The idea is that organisations collect data on the costs of previous projects, along with some basic metrics (e.g. their size in terms of lines of code). This then forms the basis for a relation between size and cost. Accordingly, if it is possible to predict the size of future projects, their costs can be extrapolated from the existing data. These approaches tend to require some parametrisation – e.g. the predicted size, and perhaps other parameters to capture the changeability of requirements or the experience of the team.
- **COCOMO is a cost prediction approach that provides pre-computed parameters.** The idea is that any organisation can calculate the predicted cost for a project by simply 'plugging in' a set of parameters that characterise the nature of the project and organisation within which it is developed.
- **Planning-poker is an example of a cost-prediction approach that does not take a 'model-based' view of cost prediction.** It is a variant of the Wideband-Delphi method, popularised by Barry Boehm. The idea is that a group of developers collaboratively estimate the expected cost of individual activities (drawing upon their own experience).
- **The accuracy of a prediction approach invariably depends on the amount of intelligence available.** At the very beginning of a project, there is little experience or knowledge to draw upon, which means that any prediction is necessarily accompanied by a great deal of uncertainty. This uncertainty diminishes as the project progresses. This was visualised by Boehm in the 'Cone of Uncertainty'.
- **It is important to keep track of progress, so that any deviation from predicted effort / cost can be detected.** This can be achieved with the use of Burndown Charts and Team Velocity.