CHAPTER 21

Managing Object-Oriented Projects

OBJECTIVES

In this chapter you will learn

- how to manage iterative projects
- how to use project planning techniques
- the main features of DSDM and XP
- what metrics are available for object-oriented projects.

21.1 Introduction

Information systems development is a complex activity that requires careful management. We discussed the need for various models of the systems life cycle in Chapter 3 and the way in which they bring structure to a project. The discussion of the UML techniques in this book highlights the need to plan and manage the whole process. There are inter-dependencies between the artefacts of software development, and their production has to be co-ordinated if the process is to be efficient. A large software development project may involve many developers, some with specialized skills. The specialist in requirements capture is required early in the project, the expert in ODBMS implementation is needed during design and construction, and the installation and support teams become involved to some extent during design and construction and more fully when the information system is complete. The different activities may require different resources whose availability has to be planned. As suggested in Chapter 2, the timing of the installation may be critical for the success of the project. The management process is further complicated by the fact that the sequence of some activities may be significant. For example, testing of a system can only begin when at least some elements have been constructed, though of course test scripts and test harnesses may be prepared early in the project. Resource allocation and project planning are considered in Section 21.2.

Iterative development approaches present particular challenges. One critical issue is how to control the number of iterations. This was discussed briefly in Chapter 3 and we examine the question in more detail in Sections 21.3, and also in 21.4 where we introduce the Dynamic Systems Development Method (DSDM). Extreme Programming (XP) is another interesting approach that is highly iterative and this is discussed in Section 21.5.

A significant part of project planning is the determination of the resources required for each stage in the project. It is also important to establish the period for which each resource will be required. This can be based only upon an estimate of how long each task is likely to take. Estimates of task duration rely heavily upon experience but can also involve the measurement of various aspects of the proposed system. Many factors affect the length of time that it takes to complete a given activity and it is not possible to quantify them all. Some factors are measurable (at least to some extent): for example, project risk, system size and complexity. Less tangible aspects of the project such as staff expertise, team morale or the difficulty of using new technology are harder to measure. Section 21.6 introduces software metrics that can be used to estimate the influence that project characteristics have upon overall development time.

The patterns that were introduced in Chapter 8 related to analysis and those discussed in Chapter 15 are mainly useful in a design context. Other patterns have been developed that are applicable to the management of the development process. In Section 21.7 we briefly introduce the idea of development organization patterns. One of the difficulties of introducing object-oriented development to an organization is that, initially at least, some systems are based on an object-oriented approach while others are not. In Section 21.8 we outline some of the problems to which this gives rise.

The move towards object-orientation introduces a series of challenges for an organization. Staff must be trained, and existing software systems must be integrated with the new approach. Any major change, and a change in development method is only one example, requires careful planning in order to minimize the attendant risks (Section 21.9).

21.2 Resource Allocation and Planning

Given the complexity of project management there is a need for tools and techniques to support the process. Yourdon (1989) identifies three particular areas of the management of software development where modelling techniques can play a useful role:

- in the estimation of money, time and people required;
- in assisting the revision of these estimates as a project continues;
- in helping to track and manage the tasks and activities carried out by a team of software developers.

Many tools (e.g. CA SuperProject) have been developed to support the management of any type of project, not just those that are focused on software development.

21.2.1 Critical Path Analysis

The technique known as Critical Path Analysis (CPA) was developed in the late 1950s for use on major weapons development projects for the US Navy (Whitten, Bentley and Barlow, 1994). Originally known as Project (or Program) Evaluation and Review Technique (PERT)[1], it is also called Network Analysis and it has been widely used on many different types of project.

For the purposes of carrying out a critical path analysis, a project is viewed as a set of activities or tasks, each of which has an expected duration. Completion of an activity corresponds to a milestone or event for the project. Each milestone also represents the start of activities that are directly dependent on the completion of the predecessor or predecessors. CPA is based on an analysis of sequential dependencies among the activities, and uses the expected duration for each task to derive an estimate of the overall duration of the project. In particular, it identifies any inter-task dependencies that are critical to the project duration—collectively these are known as the *critical path*. The preparation of a CPA chart involves the following steps.

List all project activities and milestones

A sample list for the development of the Agate Campaign Management system is shown in Figure 21.1. Each activity is labelled with a letter and has a short description. The third column in the table contains a milestone number that represents the completion of that activity.

Determine the dependencies among the activities

Some activities cannot start until another (sometimes more than one) has been completed. The preceding activities are listed in column 4. For example, in Figure 21.1 the activity `Review use cases` must be completed before the activity `Identify classes` can begin.

Activity	**Description**	**Milestone**	**Preceding activities**	**Expected duration**	**Staffing**
A	Interview users	2	–	5	2
B	Prepare use cases	3	–	2	See A
C	Review use cases	4	A,B	2	3
D	Draft screen layouts	5	C	2	2
E	Review screens	6	D	2	2
F	Identify classes	7	C	2	3
G	CRC anlaysis	8	F	4	3
H	Prepare draft class diagram	9	F	5	3
I	Review class diagram	10	G,H	4	4

Figure 21.1 Project activity table for Agate.

Estimate the duration of each activity

There are several different approaches to this, due to the uncertainty involved in estimating task duration. One that is used widely is given by the following formula:

$$ED = \frac{MOT + (4 \times MLT) + MPT}{6}$$

where *ED* is the expected duration of a task, *MOT* is the most optimistic time, *MLT* is the most likely time and *MPT* is the most pessimistic time for its completion. The *ED* is thus a weighted average of the three estimates. Each *MOT* assumes that a task will not be delayed, even by likely events such as employee absence. The *MPT* assumes that most things that can go wrong will go wrong, and that completion of the activity will be delayed to the maximum plausible extent. Equipment will arrive late, technical problems will occur and some staff will be ill. (It would be unrealistic to take this to the extreme, for example, to assume that all development staff will be struck down by an influenza epidemic—estimates should be realistic.) The *ED*s are entered in the fifth column of the table. Note that in this example the staff requirements for activities A and B have been treated as one since the two activities are highly interdependent.

Draw the CPA chart

Two main styles of notation are used for CPA charts, known respectively as ‘activity on the node’ and ‘activity on the arrow’ diagrams. Both show the same information, but they look very different from each other. To avoid confusion we present only ‘activity on the arrow’ notation. In this style each milestone is represented by a circle divided into three compartments. One compartment is labelled with the milestone number, and the other two will hold the earliest start time (EST) and the latest start time (LST) (these terms are explained below) for all activities that begin at that milestone. Figure 21.2 illustrates the notation.

The first draft of a CPA chart shows dependencies between activities and their expected durations, since this information is known or can be estimated before the diagram is drawn. The partially completed CPA chart in Figure 21.3 represents graphically the activity precedences listed in table form in Figure 21.1.

Note the *dummy activity* between milestones 8 and 9 (there are others between 2 and 3 and between 6 and 10). The explanation for this is that activities H and G both depend on milestone 9 (the completion of activity F), and are also both predecessors to milestone 10 (where activity I begins). Since H and G may not necessarily finish at the same time, an extra milestone is needed for one of these events. In effect, milestone 8 represents the completion of G, regardless of whether or not H has finished (it is not significant which of the two activities is chosen to terminate at the extra milestone). A dummy activity (with ED of 0) then needs to be introduced in order to connect milestone 8 to milestone 9, thus preserving the sequence of dependencies.

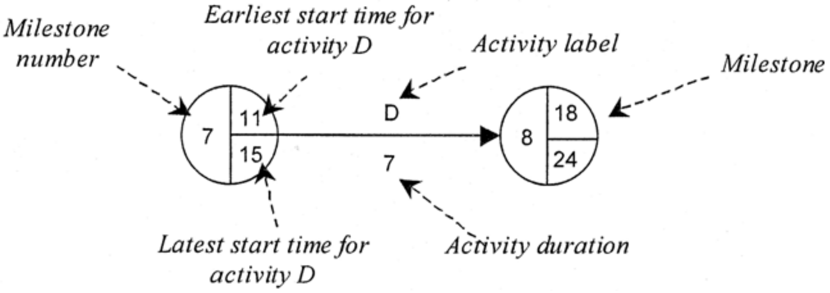

Figure 21.2 CPA notation.

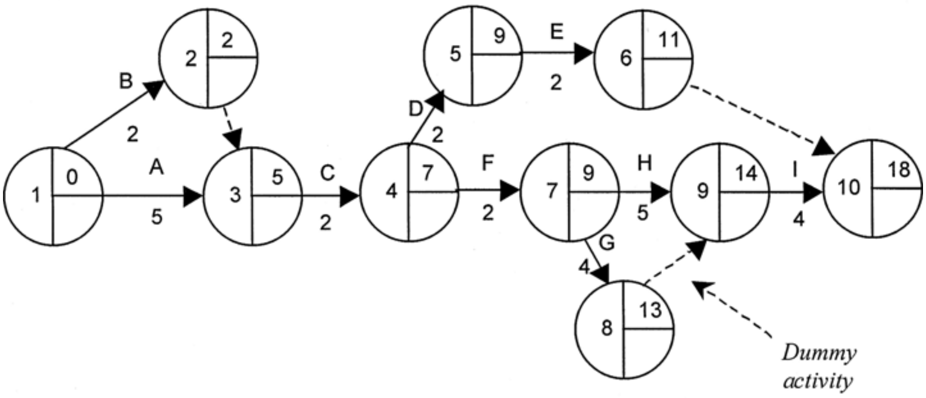

Figure 21.3 Partial CPA chart for the Agate advertising sub-system project.

The next step is to enter an earliest start time for each milestone. This is done by working through the diagram from the very first milestone to the very last (sometimes called a *forward pass*). It is a convention that the EST for the first milestone is set to 0 (elapsed time is usually measured in days, but any other unit of time can be used equally well). The EST for most other milestones is calculated simply by adding the EST of the immediately preceding activity to its duration. For example, activity C (the immediate predecessor for D) has an EST of 5 and an ED of 2, therefore the EST for activity D is 7. Where an activity has two or more predecessors, its EST is determined by the predecessor that has the latest completion time. For example, activity I is dependent upon the completion of both G and H, so its EST is set to the later of the two calculations. In general the EST for any milestone is set to the earliest time that *all* predecessor activities can be completed.

The next step requires completing the latest start time for each milestone. The latest start time is entered by working back from the last milestone (sometimes this is called a *backward pass*). The LST for the last milestone is set equal to its EST. Each preceding LST is then calculated as follows. The LST for most milestones equals the LST for its successor minus the ED of the intervening activity. For example, the LST for milestone 9 is 18 – 4 = 14 (the LST for milestone 10 minus the ED for activity I). Milestone 7 presents more of a problem as this has two successor activities, G and H. In such cases, two calculations are performed (or more, if there are more than two successors) and the earlier of the two answers is taken. For example, if the LST for milestone 7 were determined purely by activity G, this would give 14 – 4 = 10 (the LST for milestone 8 is 14). This would mean that activity G could afford to begin as late as time 10 without delaying any other activities. But a similar calculation for activity H gives 14 – 5 = 9. This means that if H begins later than time 9, its completion will be delayed beyond time 14, which in turn would delay milestones 9 and 10 and thus also activity I and the project completion. The LST for milestone 7 is therefore set to 9. In general, the LST for a milestone is set to the latest time that allows *every* activity that begins at that milestone to be completed by the LST for its succeeding milestone.

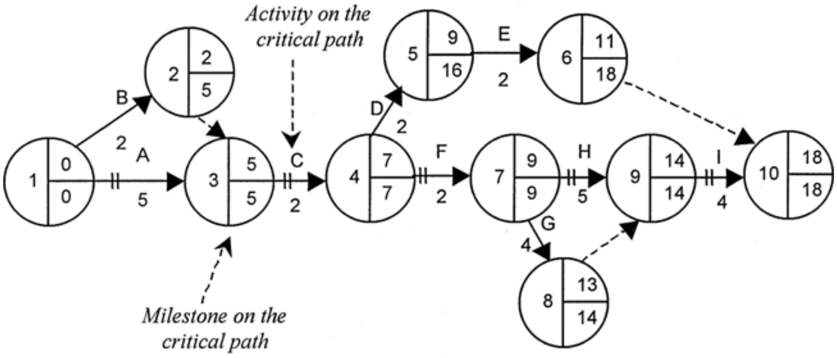

Figure 21.4 The critical path for the Agate advertising sub-system project.

Identify critical path

Once all LSTs have been entered onto the diagram, the *slack time* (or *float*) for each activity can be calculated. This is the difference between an activity's EST and its LST, and it represents the time by which that particular activity can be delayed without affecting the overall duration of the project. The path through all milestones that have a slack time of 0 is called the *critical path*. This is indicated by a double bar across activity arrows that connect the milestones. These milestones, and their intervening activities, are critical to the completion of the project on time. Milestones that have an EST that is different from their LST are not critical, in the sense that they have some scheduling flexibility. The completed diagram is shown in Figure 21.4.

A CPA chart is an effective tool for identifying those activities whose completion is critical to the completion of a project on time. If any activity that is on the critical path falls behind schedule then the project as a whole is behind schedule. However, while critical activities naturally receive the closest scrutiny, all project activities should be monitored. Delay even in a non-critical activity can, if it is sufficiently severe, alter the critical path.

CPA charts have limitations, chief among which is that they are not very useful for representing the extent of any overlap between activities. As a result, they are often used in conjunction with other techniques, particularly the Gantt chart (described in the following section).

21.2.2 Gantt charts

The Gantt chart (named for its inventor Henry Gantt) is a simple time-charting technique that uses horizontal bars to represent project activities. The horizontal axis represents time and is often labelled with dates or week numbers so that the completion of each activity can be monitored easily. Activities are listed vertically on the left of the chart, and the length of the bar for each activity corresponds to its ED.

A Gantt chart shows the overlap of activities clearly and this provides an effective way of considering alternative resource allocations. The Gantt chart can be drawn with either dashed lines or dashed boxes that show the slack time for non-critical activities.

A Gantt chart is also easy to convert into a stacked bar graph that can be used to show the way that the total resource allocation for a project changes over time. Figure 21.5 shows a Gantt chart and a staffing bar chart for the Agate advertising subsystem project. The staffing chart is derived as follows. The final column of Figure 21.1 gives a staff allocation for each activity. The Gantt chart is read vertically for each successive time interval to calculate the total number of staff required for all project activities combined. The result is shown as a vertical bar that indicates the total staffing required at that time.

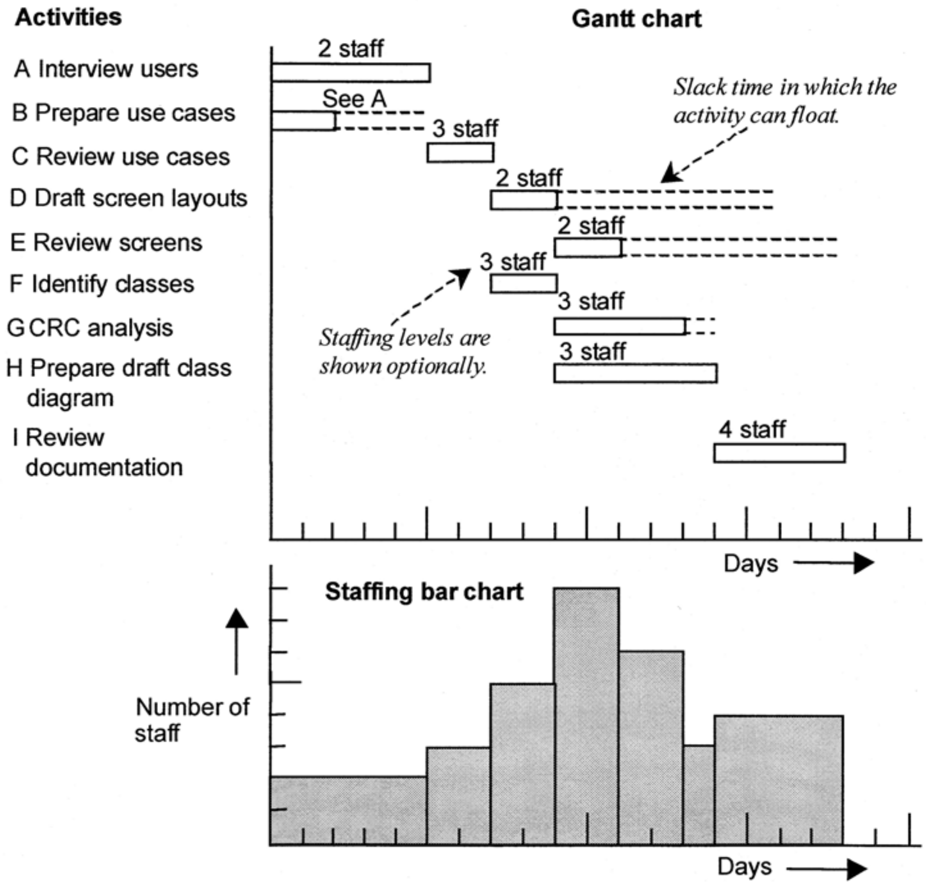

Figure 21.5 Gantt Chart with staffing bar chart.

Activities that are not on the critical path can have their start time adjusted. For example, activity E cannot begin until time 9 at the earliest, but it could start as late as time 16 without affecting project completion. A project manager can adjust the resource profile to accommodate staff availability, a process known as resource *smoothing*. For example, activities E, G and H can occur concurrently. H is on the critical path and cannot be moved, but the slack time for E allows it to begin at time 16 instead of time 9. The manager can minimize the overall resource requirement by rescheduling activity E to begin at time 14. This should be done with care, however, as it may move an activity onto the critical path. Figure 21.6 shows the smoothed resource profile.

The Gantt chart is a useful tool for resource monitoring. The progress of each activity can be shown independently and compared against planned progress. In Figure 21.6 the Gantt chart reflects the current state of a project. Activities A, B, C and F are complete. D has not been started and will shortly become critical. G is on schedule but H is behind schedule. Activities D and H need to be investigated by the project manager. Possible reasons for the delay include:

- an unexpected technical problem;
- staff absence;
- the complexity of the activity has been underestimated.

The project manager must decide how best to get the project back on schedule. Perhaps the allocation of more staff to activity H would resolve the difficulty, but throwing staff at an activity can be counter-productive for several reasons. First, if the additional staff are not familiar with the activity, they may require extensive briefing, thus reducing the time that is available to complete the work. Second, as team size increases so do the communication overheads. Third, some activities are limited in the maximum number of staff that can be involved. For example, a car repair may require 20 person-hours of work to complete. This does not mean that 20 mechanics working together could do the work in one hour. It is more likely that only, say, five mechanics could work productively on one car at any one time. The best possible repair time would therefore be four hours.

When a critical path activity is behind schedule it may not be possible to regain the lost time. A project manager then has really only two options, and ideally the choice should be discussed with the client. First, the project deadline can be moved to accommodate the delay. Second, the scope of the project or the quality of the product can be reduced to permit completion on time. The last approach requires an analysis of all uncompleted critical path activities in order to identify what can be omitted to reduce their EDs. The resultant changes may alter the critical path, and activities whose completion was not critical before may now become critical. Any attempt to reduce the scope of a project requires user involvement to ensure that only non-critical features are omitted, particularly during the first increment.

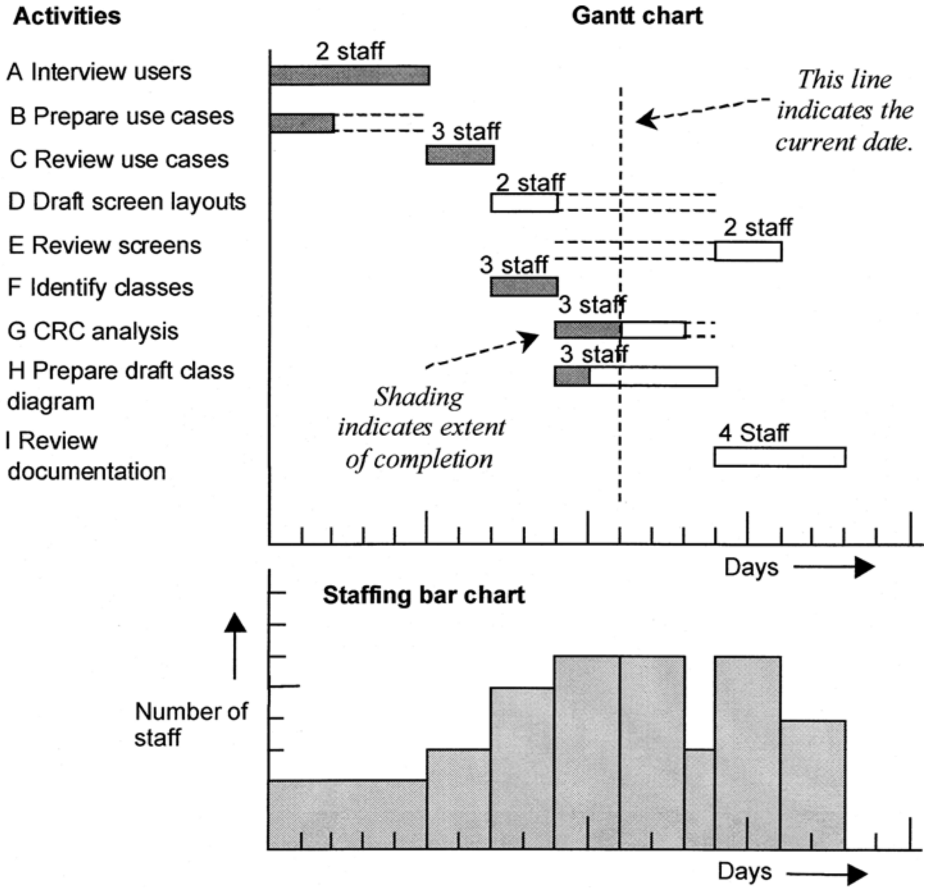

Figure 21.6 Gantt chart showing activity completion and smoothed staff profile.

The process of planning a project is iterative. An alteration to the staff allocation for an activity will probably change its ED, which in turn may change the critical path. A project manager must operate within the constraints of staff and resource availability.

21.3 Managing Iteration

In Chapter 3 we considered prototyping as an iterative development activity and discussed the need to specify objectives for the prototyping activity so that they can provide criteria to control the number of iterations (Figure 3.6 shows an example of an iterative life cycle). At the end of an iteration the product, say a prototype, is evaluated against pre-defined objectives. In general terms this seems straightforward but in practice it can be difficult to determine whether the objectives of the iterative activity have been achieved.

Let us suppose that the interface for the Agate Campaign Management system is to be developed by prototyping, with the explicit objective of producing an interface with which the campaign staff are happy. However, although this objective may be worthwhile it is of little use for the management of the activity. Imagine that the users are never completely happy at the end of any iteration: they will continue to suggest further improvements without end. As the process continues, the nature of the modifi- cations that are suggested at each iteration will change. Over time the improvements will become cosmetic and ultimately peripheral to the utility of the system. It would be sensible to end the iterative process before this point is reached. A more suitable objective for the exercise might be phrased as follows:

Continue the iterations until fewer than five cosmetic changes are requested on a single iteration.

It is still not clear how many iterations will be needed to satisfy this criterion. If the project has unlimited time and an unlimited budget this may not be a problem but this is unlikely to be the case. Additional criteria can be added to tighten up the objectives, such as the following.

The prototyping phase must be completed before the end of October and must not exceed 50 developer-hours.

21.4 Dynamic Systems Development Method

The Dynamic Systems Development Method (DSDM) is a management and control framework for rapid application development (RAD). The distinction between RAD and prototyping is sometimes blurred. A RAD approach aims to build a working system rapidly while a prototyping approach also builds rapidly, but usually only produces a partially complete system, typically to confirm some aspect of the requirement. Because both approaches aim to build software quickly, similar development environments are used and one approach to prototyping continues the development of a prototype incrementally until it becomes a working system. In effect this is a RAD development approach.

The traditional waterfall approach to systems development has deficiencies, particularly the time taken to deliver a working system and the inflexibility of the approach to requirements change. Iterative approaches to development can also be problematic. As suggested above it is sometimes difficult to cease the iterations when they become unproductive. In the early 1990s RAD became much more popular and was viewed as a way of matching systems development to the fast changing needs of business. However, there were until recently no commonly accepted structures for either the use or the management of RAD. The DSDM consortium was formed in 1994 to produce an industry standard definition of the RAD process and DSDM was subsequently defined. The DSDM framework defines structure and controls to be used in a RAD project but does not specify a development methodology. DSDM may be used with either an object-oriented or a structured methodology. DSDM takes a fundamentally different perspective on project control. Rather than viewing requirements as fixed and attempting to match resources to the project, DSDM fixes resources for the project, fixes the time available and then sets out to deliver only what can be achieved within these constraints.

DSDM is based upon nine underlying principles (Stapleton, 1997):

- Active user involvement is imperative. Many other approaches effectively restrict user involvement to requirements acquisition at the beginning of the project and acceptance testing at the end of the project. In DSDM users are members of the project team and include one known as an ‘Ambassador’ user.
- DSDM teams are empowered to make decisions. A team can make decisions that refine the requirements and possibly even change them without the direct involvement of higher management.
- The focus is on frequent product delivery. A team is geared to delivering products in an agreed time period and it selects the most appropriate approach to achieve this. The time periods are known as timeboxes and are normally kept short (2 to 6 weeks). This helps team members to decide in advance what is feasible. Products can include analysis and design artefacts as well as working systems.
- The essential criterion for acceptance of a deliverable is fitness for business purpose. DSDM is geared to delivering the essential functionality at the specified time.
- Iterative and incremental development is necessary to converge on an accurate business solution. Incremental development allows user feedback to inform the development of later increments. The delivery of partial solutions is considered acceptable if they satisfy an immediate and urgent user need. These solutions can be refined and further developed later.
- All changes during development are reversible. If the iterative development follows an inappropriate development path then it is necessary to return to the last point in the development cycle that was considered appropriate. Changes are limited within a particular increment.
- Requirements are initially agreed at a high level. Once requirements are fixed at a high level they provide the objectives for prototyping. The requirements can then be investigated in detail by the DSDM teams to determine the best way to achieve them. Normally the scope of the high level requirements is not changed significantly.
- Testing is integrated throughout the life cycle. Since a partially complete system may be delivered it must be tested during development, rather than after completion. Each software component is tested by the developers for technical compliance and by user team members for functional appropriateness.

- A collaborative and co-operative approach between all stakeholders is essential. The emphasis here is on the inclusion of all stakeholders in a collaborative development process. Stakeholders not only include team members, but others such as resource managers and the quality assurance team.

21.4.1 The DSDM life cycle

The DSDM life cycle has these phases:

- feasibility study,
- business study,
- functional model iteration,
- design and build iteration,
- implementation.

The relationships between the phases are shown graphically in Figure 21.7, and each is described in more detail below. Note that the last three are actually iterative processes, but for the sake of clarity this is not shown explicitly in Figure 21.7.

The *feasibility study* phase determines whether the project is suitable for a DSDM approach. It typically lasts only weeks, whereas the feasibility stage can last months on a traditionally run project. The study should also answer questions such as the following:

- Is the computerized information system technically possible?
- Will the benefit of the system be outweighed by its costs?
- Will the information system operate acceptably within the organization?

The *business study* phase identifies the overall scope of the project and results in agreed high level functional and non-functional requirements. Maintainability objectives are set at this stage and these determine the quality control activities for the remainder of the project. There are three levels of maintainability:

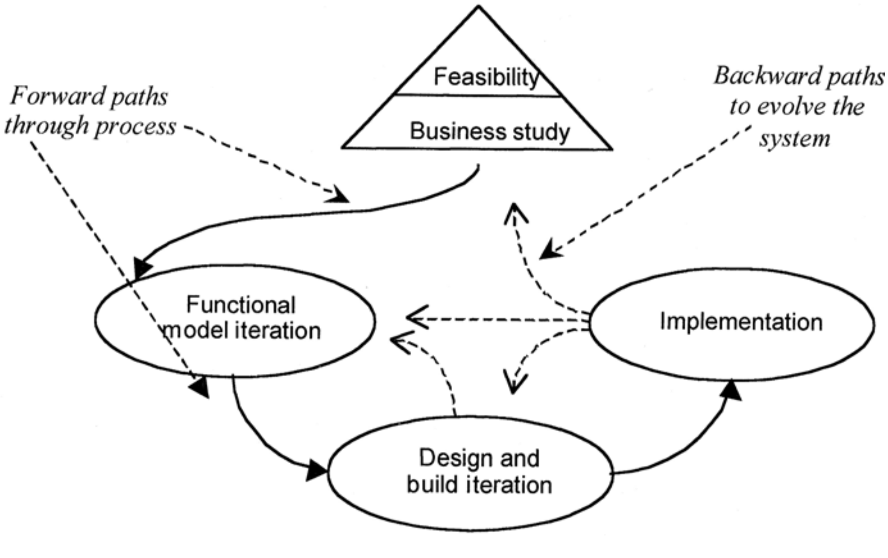
Feasibility
Business study
Forward paths through process
Backward paths to evolve the system
Functional model iteration
Implementation
Design and build iteration

Figure 21.7 Simplified DSDM life cycle (adapted from Stapleton, 1997).

- maintainable from initial operation;
- not necessarily maintainable when first installed but this can be addressed later;
- short life-span system that will not be subject to maintenance.

Where the third level of maintenance is chosen care should be taken to ensure that the system is discontinued at the end of its allotted time, otherwise it may become subject to maintenance requests that are difficult to service.

The *functional model iteration* phase is concerned with the development of prototypes to elicit detailed requirements. The intention of DSDM is to develop prototypes that can ultimately be delivered as operational systems, so these must be built to be sufficiently robust for operational use and also to satisfy any non-functional requirements such as performance. When completed the functional model comprises high level analysis models and documentation together with prototypes that are concerned with detailed functionality and usability. During the functional model iteration the following activities are undertaken:

- the functional prototype is identified;
- a schedule is agreed;
- the functional prototype is created;
- the functional prototype is reviewed.

The *design and build iteration* phase is concerned with developing the prototypes to the point where they can be used operationally. The distinction between the functional model iteration and the design and build iteration is not clear-cut and both phases can run concurrently. The activities for the design and build iteration phase are very similar to those described above for the functional model iteration phase.

The *implementation* phase deals with the installation of the latest increment including user training. At this point it is important to review the extent to which the requirements have been met. If they have been fully satisfied the project is complete. If some non-functional requirements have yet to be addressed the project may return to the design and build iteration phase. If some element of functionality was omitted due to time constraints the project may return to the functional model iteration phase. If a new functional area is identified the project may return to the business study phase. The return flows of control are shown with dashed arrows in Figure 21.7. Implementation comprises the following iterative activities:

- producing user guidelines and gaining user approval;
- training users;
- implementing the system;
- reviewing the business requirements.

21.4.2 Timeboxing

Timeboxing is an approach for fixing the resource allocation for a project or a part of a project. It limits the time available for the refinement of requirements, design, construction and implementation as appropriate. A RAD project has a fixed completion date that defines an overall timebox for the project. A DSDM approach to project management will then identify smaller timeboxes within this, each with a set of prioritized objectives. Each timebox produces one or more deliverables that allow progress and quality to be assessed. Within a timebox the team have three major concerns. They must first carry out any investigation needed to determine the direction that should be taken for that part of the project. They must then develop and refine the specified deliverables. Finally they must consolidate their work prior to the final deadline.

It is sometimes difficult to prioritize the requirements that will be actioned during a timebox. One way of doing this is to apply the set of rules that are known as the *MoSCoW* rules (for Must … Should … Could … Want).

Must have requirements are crucial. If these are omitted the system will not operate. In DSDM the set of *Must have* requirements are known as the minimum usable subset.

Should have requirements are important but if necessary the system can operate usefully without them.

Could have requirements are less important and provide less benefit to the user.

Want to have but will not have this time around requirements can reasonably be left for development in a later increment.

All of these requirements are important for the final system but not to the same extent. If the full set cannot be addressed within a timebox, the MoSCoW categorization can be used to focus the requirements in an appropriate way.

21.5 Extreme Programming

Extreme Programming (XP) is a novel combination of elements of best practice in systems development. It was first publicized by Kent Beck (Beck, 2000) and incorporates a highly iterative approach to development. It has become well known in a relatively short period of time for its use of *pair programming* though it encompasses various other important ideas. Pair programming involves writing the program code in pairs and not individually. At first sight it would appear that this approach would significantly increase the staffing level and hence the cost of developing an information system but the advocates of XP claim otherwise.

Beck (2000) identifies the four underlying principles of XP as communication, simplicity, feedback and courage.

Communication. Poor communication is a significant factor in failing projects, XP highlights the importance of good communication among developers and between developers and users.

Simplicity. Software developers are sometimes tempted to use technology for technology's sake rather than seeking the simplest effective solution. Developers justify complex solutions as a way of meeting possible future requirements. XP focuses on the simplest solution for the immediate known requirements.

Feedback. Unjustified optimism is common in systems development. Developers tend to underestimate the time required to complete any particular programming task. This results in poor estimates of project completion, constant chasing of unrealistic deadlines, stressed developers and poor product quality. Feedback in XP is geared to giving the developers frequent and timely feedback from users and also in terms of test results. Work estimates are based on the work actually completed in the previous iteration.

Courage. The exhortation to be courageous urges the developer to throw away code that is not quite correct and start again rather than trying to fix the unfixable. Essentially the developer has to leave unproductive lines of development despite personal investment in the ideas.

XP also argues that embracing change is important and key to systems development and that development staff are motivated by producing quality work.

Requirements capture in XP is based on *user stories* that describe the requirements. These are written by the user and form the basis of project planning and the development of test harnesses. User stories are very similar to use cases though some proponents of XP suggest that there are key differences in granularity. A typical user story is about three sentences long and does not include any detail of technology. When the developers are ready to start writing the system they get detailed descriptions of requirements face to face with the customer. Beck (2000) talks about the systems development process as being driven by the user stories in much the same way that the USDP is use case driven. XP involves the following activities.

- *The planning game* involves quickly defining the scope of the next release from user priorities and technical estimates. The plan is updated regularly as the iteration progresses.
- The information system should be delivered in *small releases* that incrementally build up functionality through rapid iteration.
- A unifying *metaphor* or high level shared story focuses the development.
- The system should be based on as *simple design*.
- Programmers prepare unit *tests* in advance of software construction and customers define acceptance tests.
- The programme code should be restructured to remove duplication, simplify the code and improve flexibility—this is known as *refactoring*, and is discussed in Fowler (1999) in detail.
- Pair programming means that code is written by two programmers using one workstation.
- The code is owned collectively and anyone can change any code.
- The system is integrated and built frequently each day. This gives the opportunity for regular testing and feedback.
- Normally staff should work no more than forty hours a week.
- A user should be a full-time member of the team.
- All programmers should write code according to agreed standards that emphasize good communication through the code.

The XP approach is best suited to projects with a relatively small number of programmers—say no more than ten. In XP it is critical to maintain clear communicative code and to have rapid feedback. If a project precludes either of these then XP is not the most appropriate approach. One key feature of XP is that the code itself is the design documentation. This runs counter to some aspects of the approach suggested in this book. We have suggested that requirements are effectively analysed and suitable designs produced though the use of visual models using UML. Nonetheless XP does offer interesting insights into how software development projects can be organized and managed.

21.6 Software Metrics

Planning and managing a software development project requires the estimation of the resources required for each of its constituent activities. A resource estimate for an activity can be based upon subjective perceptions of the activity or it can be based upon measurements of size and complexity, either of the activity itself or of the artefact that is produced. DeMarco's (1982) frequently quoted aphorism sums it up:

> You can't control what you can't measure.

We are used to applying measures in many forms of human endeavour. For example, the productivity of a car factory can be measured quite accurately in terms of the time taken to construct a car. The use of metrics in software development is still rather limited. The term software engineering was coined in the 1960s as part of an attempt to introduce the greater degree of rigour that was seen in the management of other types of engineering. It is a feature of most engineering disciplines that careful measurement is used to assess the efficiency and effectiveness of artefacts and of processes. A *software metric* is a measure of some aspect of software development, either at project level—usually its cost or its duration—or at the level of the application—typically its size or its complexity.

Software metrics can be divided broadly into two categories: *process metrics* that measure some aspect of the development process and *product metrics* that measure some aspect of the software product. (We use the term software product broadly to include anything that is produced during a software development project, including for example analysis models and test plans as well as program code.) Two examples of process metrics are the project cost to date and the amount of time spent so far on the project. Product metrics relate to the information system that is under development. One of the simplest product metrics is the number of classes in an analysis class diagram.

Software metrics can also be categorized as *result* or *predictor* metrics, which are used respectively to measure outcomes and to quantify estimates. The current cost of a project is a result metric (even though this is only an interim outcome that will probably be modified tomorrow). A measure of class size (a crude measure might be a simple count of attributes and operations) would be a predictor metric, so called because it can be used as a basis for predicting the time that it will take to produce program code for that class. Result metrics are also known as control metrics (Sommerville, 1992) since they are used to determine how management control should be exercised. For example, a measurement of the current level of progress in the project is used to decide whether action is necessary to bring the project back onto schedule. The term 'predictor metric' is generally applied only to a measure of some aspect of a software product that is used to predict some other aspect of the product or of the project progress. Predictor metrics are not used solely for estimation. The results obtained from their application to a project may indicate, for example, that the system will be difficult to maintain or that it may offer very low levels of reuse. Since neither outcome is desirable, managers may attempt to change the design for the system or the process for its development in order to improve the system.

The validity of predictor metrics is based upon three assumptions, often made only tacitly by managers (Sommerville, 1992):

- there is some aspect of a software product that can be accurately measured;
- there is a relationship between the measurable aspect and some other relevant characteristics of the product;
- this relationship has been validated and can be expressed in a model or a formula.

The last of these assumptions suggests that a significant volume of historical data must be collected so that an appropriate statistical analysis can validate the relationship. In practice this is only feasible if the data collection is automated (for example, it is done by a CASE tool). Generally speaking, software developers do not regard the capture of metrics data as an important part of their role. Their attention is inevitably focused on the delivery of the product on time. Another factor that limits the uptake of metrics is the possibility of using them to monitor the performance of the developers themselves. This may give rise to concern among the developers about how data on their performance might be used. This is of course an ethical issue as well as a management one.

A number of metrics have been identified for use with structured analysis and design approaches. For example, De Marco (1982) developed a complexity metric known as the Bang Metric for use on structured analysis projects. Other metrics that focused on the degree of coupling and cohesion between program modules have also been suggested for use with a structured design approach. Most of these metrics were used little despite the introduction of automated CASE support.

A number of authors have identified metrics for object-oriented systems development (Lorenz, 1994; de Champeaux, 1997; Graham, 1995). For instance de Champeaux suggests a list of desirable features as part of a general description of a useful metric:

- either it is elementary and focuses on a single well-defined aspect, or it is an aggregation of elementary metrics;
- it is suitable for automated evaluation;
- gathering the metric data is not too costly;
- it is intuitive;
- application to a composite is equivalent to application to the components and summation of the individual results;
- the metric can be measured numerically and arithmetic operations are meaningful.

He lists a series of quality metrics, for example, a dependency metric that provides a measure of the stability of a system. When applied to a package or a sub-system this measures the degree of inter-package or inter-sub-system coupling. It is calculated by the formula:

$$I = \frac{CE}{(Ca + Ce)}$$

where I is the instability of the system, Ca is the level of afferent coupling (the number of classes outside the package that depend on classes within the package), and Ce is the level of efferent coupling (the number of classes outside the package upon which classes within the package depend) (adapted from de Champeaux, 1997).

When *I* is zero the package is maximally stable and has no dependencies on classes in other packages. When *I* is 1 the package is maximally unstable and has dependencies only on classes outside itself.

The ability of a package to absorb change is reflected (in part at least) by the ratio of abstract classes to all classes within the package. Where this is zero then the package consists solely of concrete classes and is difficult to change. A ratio of one indicates the presence of no concrete classes at all and it is easier to change.

Lorenz and Kidd (1994) suggest a wide range of metrics including metrics for application size and class size. Application size is essentially determined from the number of use cases and the number of domain classes[2] together with multiplying factors that reflect the complexity of the user interface. The size of a class is determined by the number of attributes and operations it has and the size of the operations. Size metrics such as this can be used to estimate the resource requirement for a project providing that appropriate historical data is available to derive and validate the relationship.

21.7 Process Patterns

We have already discussed analysis patterns in Chapter 8 and design patterns in Chapter 15. Coplien (1995) has defined a pattern language that is focused on the development process. The pattern language comprises both organization and process patterns. As with design patterns they capture elements of experience as problem—solution pairs. The patterns address issues such as team selection, organizational size, team structure, the roles of team members and so on. Conway's Law (Pattern 14) discusses how the architecture of the system comes to reflect the organizational structure or vice versa. Mercenary Analyst (Pattern 23) is concerned with producing project documentation successfully. This pattern suggests that it is frequently more effective to hire a technical writer who can focus solely on the documentation.

21.8 Legacy Systems

There are many definitions of the term legacy system. We take it to mean any computerized information system that has been in use for some time, that was built with older technologies (perhaps using a different development approach at different times) and, most importantly, that continues to deliver benefit to the organization. Most computerized information systems interact with other computerized information systems. They may share data, the output from one may be an input to another and so on. Any new information system is likely to need to interact with older legacy systems that have not been built using the same technologies. Redeveloping legacy systems so that they interact appropriately with new systems is likely to be prohibitively expensive and probably involves too much risk. These legacy systems may be critical to the operation of the organization. The problem is one of integrating new object-oriented systems with non-object-oriented systems.

One strategy that enables the interoperation of old and new is the use of an *object wrapper* (Graham, 1995). An object wrapper functions as an interface that surrounds a non-object-oriented system so that it presents an interface suitable for use with new object-oriented systems. Essentially the old system appears to be object-oriented. The form of the wrapper depends on the nature of the legacy system. Where the old system uses text or form-based screen interfaces the wrapper may involve program code that reads data from the screen and writes data to the screen (sometimes known as screen scrapers) using some form of virtual terminal.

When an organization embarks upon object-oriented software development there may be an intention to migrate all existing software to the new technologies. The cost and risk involved may constrain this but where it is feasible to migrate systems it is important to manage the process carefully. Wrappers may be used initially to provide an object-oriented interface to the system. Then, once the system has been wrapped, it can be redeveloped (perhaps one sub-system at a time) without affecting its interface to other systems. A further variation on this approach is to use the Façade pattern (see Chapter 20) to wrap sub-systems so that they can be migrated incrementally.

21.9 Introducing Object Technology

The introduction of any new way of working requires careful planning. Introducing object technology is a very significant change. Both the approach to systems development and the development technology are new. Staff must be trained in the principles of object-orientation, in new analysis and design techniques, in one or more objectoriented programming languages, in the use of new development environments and perhaps also in how to use a particular object-oriented database management system. Almost every aspect of the software development activity will change. This is not just a matter of training the developers; they also must gain experience in applying the new techniques and using the new technology. The safest way to achieve this is to apply object-orientation within a pilot project in the first instance (as suggested for FoodCo in Box B1.2). The move to object-orientation must be carefully planned and controlled and should ideally follow the steps shown below:

- identify a suitable pilot project;

- train the relevant staff;
- monitor the project carefully when it is under way;
- review the project implementation;
- identify the lessons learned and migrate this experience to other suitable projects.

Identification of a suitable pilot project. A pilot project should ideally not be subject to very tight timescales. The team is likely to require additional time to become familiar with the concepts and notation. The project should be amenable to an objectoriented approach in terms of its application domain and implementation technology.

Training. The move to object-orientation requires a different perspective on the problem domain. Training should be planned so that developers are trained in the new techniques immediately before they need to use them. It is important that a developer20should understand the conceptual basis of object-orientation before embarking upon learning a new modelling notation (for example UML) or a new programming language (for example Java). Training and familiarization with the selected CASE toolset should be included. Users who will interact with the project should also be trained so that they can participate effectively in user reviews.

Monitoring the project. To gain maximum benefit from the move to objectorientation20it is important to monitor its use within the organization. This provides feedback on the effectiveness of the training, the most appropriate way of applying the techniques and the project management requirements.

Review project implementation. A post-implementation review provides an opportunity to determine whether or not the expected benefits of the approach have been realized. It is important not to place unreasonable expectations upon a pilot project since unfulfilled aspirations may then over-shadow its successes. For instance, the potential benefits of reuse are unlikely to be achieved immediately after an organization has migrated to object-orientation. Even with careful management, significant levels of reuse may not appear for a further two or three years.

Migrate experience to other suitable projects. The experience of the pilot project will suggest adjustments that can be made to improve the way that object-orientation is to be used in the organization. Members of the initial project team may be used to seed other projects with their experience of the use of object-oriented development methods.

21.10 Summary

Systems development projects are similar to any other project in their need for sound management to ensure that they are completed within budget and on time. Some standard project management techniques used for this purpose include Critical Path Analysis and Gantt Charts, both of which can be used for project planning, to monitor resource utilization and to monitor project progress.

Object-oriented projects generally use a rapid development approach, and DSDM has emerged as the predominant framework for RAD project management. An objectoriented approach works particularly well with DSDM. XP provides an interesting alternative approach to systems development and provides some valuable insights.

The successful management of systems development relies on the quantification of various measures of effort, progress and complexity. A number of software metrics have been developed for this purpose including some that are intended specifically for use within object-oriented development.

The introduction of object-orientation to an organization should be carefully planned and managed. It is also important to ensure that existing computerized information systems can continue to operate alongside object-oriented systems, for example by the use of object wrappers.

The success of an information systems development project depends upon many factors, but in the final analysis the most important factor by far is the skill of the development staff. Effective project management enables developers to apply their knowledge and expertise in the most productive way.

When used by suitably skilled developers and managed by appropriately qualified managers, object-oriented development offers a means for capturing, modelling and building complex information systems that fully meet the needs of their users.

Review Questions

21.1 In CPA how is the slack time for an activity calculated?

21.2 In CPA how can you decide whether a milestone is on the critical path?

21.3 What are the advantages of a Gantt chart when compared to a CPA chart?

21.4 What are the advantages of a CPA chart when compared to a Gantt Chart?

21.5 What are the main life cycle stages for DSDM?

21.6 What are the MoSCoW rules and how do they help in the management of a timeboxed activity?

21.7 What are the underlying principles of XP?

21.8 What are the key elements of the XP approach?

21.9 How do object wrappers help integrate new object-oriented systems with existing nonobject-oriented systems?

21.10 What factors should be considered when migrating to object technology?

Case Study Work, Exercises and Projects

21.A Prepare a CPA chart, a Gantt chart and a staffing profile for the project activities shown in the table at the foot of this page.

21.B FoodCo had never used object-oriented development methods until the start of the Production Costing System project. Identify other projects at FoodCo that would be alternative candidates for a pilot project, when considered as the first step in a company-wide migration process. Compare these in terms of their appropriateness.

21.C Critically evaluate a project management software package that is available to you. What changes to its functionality and non-functional characteristics would be required for it to be integrated with a CASE tool that supports UML?

Activity	Description	Milestone	Preceding activities	Expected duration	Staffing
A	Interview users	2		4	3
B	Prepare use cases	3		3	2
C	Review use cases	4	A, B	3	3
D	Draft screen layouts	5	C	4	2
E	Review screens	6	D	2	2
F	Identify classes	7	E	2	3
G	CRC analysis	8	F	3	2
H	Prepare draft class diagram	9	F	4	3
I	Review documentation	10	G, H	4	4

Further Reading

- De Marco (1982) provides a good introduction to the issues of managing software projects although the development techniques discussed are not object-oriented.
- Stapleton (1997) is a comprehensive and readable introduction to DSDM. Information is also available at www.dsdm.org. Newkirk and Martin (2001) provide a useful description of applying XP to a project. Further information about XP can also be found at www.xprogramming.com. The integration of XP and the Rational Unified Process is discussed in a white paper by Gary Pollice (2001) on the Rational website www.rational.com.
- Graham (1995) examines the migration to object technology and is well worth reading.
- Lorenz and Kidd (1994), Graham (1995) and de Champeaux (1997) address the use of objectoriented metrics. Melton (1995) is a collection of research articles on the subject, the first chapter being an interesting history of the development of software metrics.
- Coplien's pattern language (Coplien, 1995) describes many interesting patterns that will be familiar to project managers, albeit in many different guises.

---

[1] Strictly speaking PERT is more elaborate than CPA, using statistical measures in addition to critical path analysis, but the terms are generally used synonymously. CPA charts are sometimes known as activity diagrams (Skidmore et al., 1994) but this name risks confusion with UML activity diagrams.

[2] Lorenz and Kidd use the terms 'scenario scripts' and 'key classes'.