4

SELECTION OF AN APPROPRIATE PROJECT APPROACH

OBJECTIVES

When you have completed this chapter you will be able to:

- evaluate situations where software applications could be acquired off-the-shelf rather than being built specially;
- take account of the characteristics of the system to be developed when planning a project;
- select an appropriate process model;
- make best use of the waterfall process model where appropriate;
- reduce some risks by the creation of appropriate prototypes;
- reduce other risks by implementing the project in increments;
- identify where unnecessary organizational obstacles can be removed by using agile development methods.

4.1 Introduction

The development of software in-house usually means that:

- the developers and the users belong to the same organization;
- the application will slot into a portfolio of existing computer-based systems;
- the methodologies and technologies are largely dictated by organizational standards and policies including the existing enterprise architecture.

However, a software supplier could carry out successive development projects for a variety of external customers. They would need to review the methodologies and technologies to be used for each individual project. This decision-making process has been called *technical planning* by some, although here we use the term *project analysis*. Other terms for this process are *methods engineering* and *methods tailoring*. Even where

Scanned by CamScanner

development is in-house, any characteristics of the new project requiring a different approach from previous projects need to be considered. A wide range of system development methods exists, but many organizations get along without using any of the recognized approaches. Where methods are used, 'means–end inversion' can happen: developers focus on the means – the procedures and intermediate products of a prescribed method – at the expense of the 'end', the actual required outcomes of the work. These issues are the subject of this chapter.

The relevant part of the Step Wise approach is Step 3: *Analyse project characteristics*. The selection of a particular process model could add new products to the Project Breakdown Structure or new activities to the activity network. This will generate inputs for Step 4: Identify the products and activities of the project (see Figure 4.1).

B. Fitzgerald, N. L. Russo and T. O'Kane (2003). 'Software development method tailoring at Motorola' *Communications of the ACM* 46(4) 65–70 provides a good insight into how method tailoring works in practice.

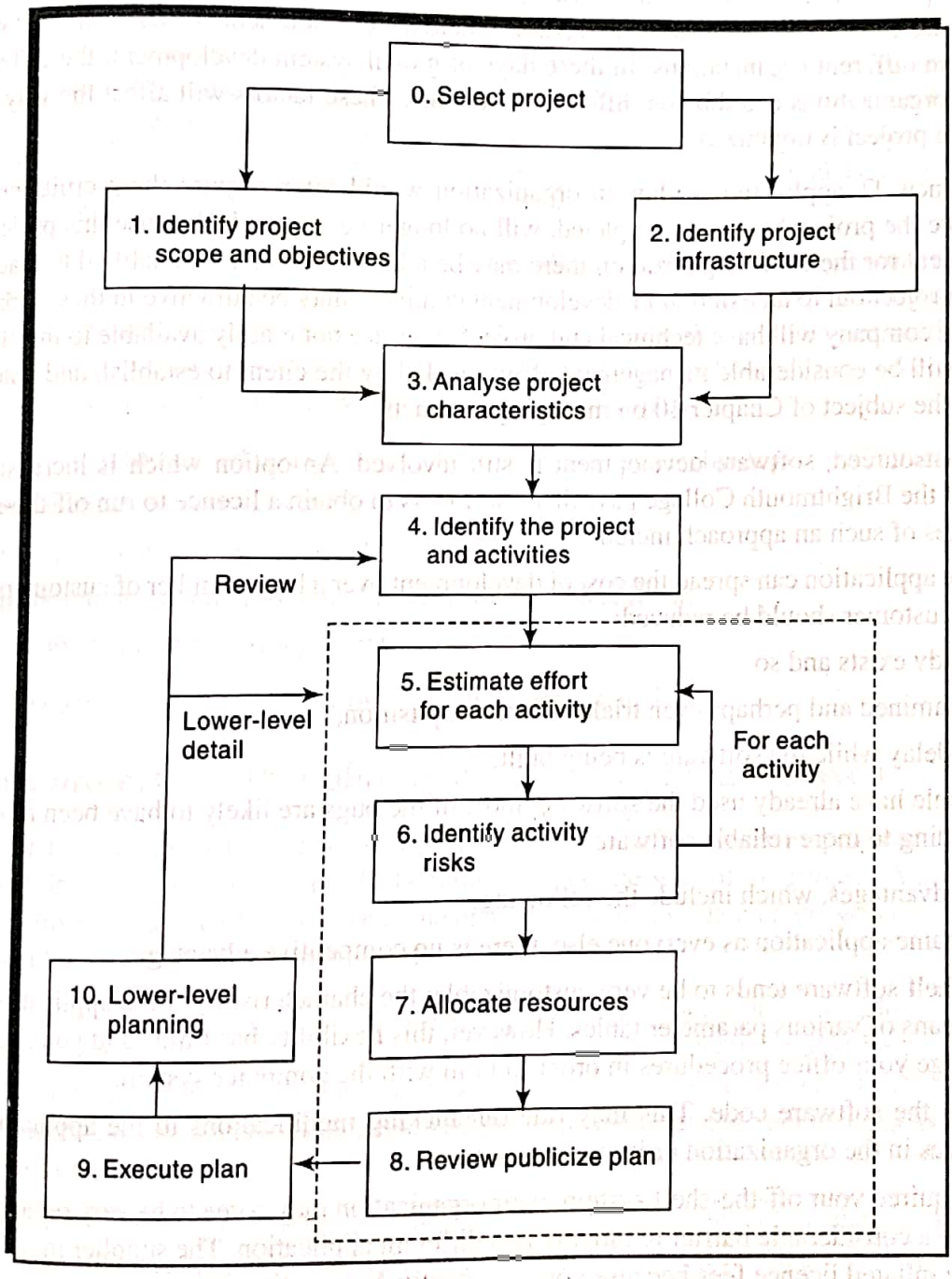

FIGURE 4.1 Project analysis is the subject of Step 3

Scanned by CamScanner

In the remainder of this chapter we will look at how the characteristics of a project's environment and the application to be delivered influence the shape of the plan of a project. We will then look at some of the most common *process models*, namely the waterfall approach, prototyping and incremental delivery. Some of the ideas of prototyping and incremental delivery have been further developed and made part of *agile methods*. We will have a look at how these *lightweight* processes have been designed to remove what have been seen as the bureaucratic obstacles created by more formal, *heavyweight* methods.

4.2 Build or Buy?

The communication challenges of geographically dispersed projects are discussed in Chapter 12 on working in teams.

Software development can be seen from two differing viewpoints: that of the developers and that of the clients or users. With *in-house development*, the developers and the users are in the same organization. Where the development is *outsourced*, they are in different organizations. In these days of global system development, the different organizations could be on different continents. These factors will affect the way that a project is organized.

The development of a new IT application within an organization would often require the recruitment of technical staff who, once the project has been completed, will no longer be required. Because this project is a unique new development for the client organization there may be a lack of executives qualified to lead the effort. Contracting the project out to an external IT development company may be attractive in these circumstances. The contracting company will have technical and project expertise not readily available to the client. However, there would still be considerable management effort needed by the client to establish and manage the contract and this is the subject of Chapter 10 on managing contracts.

Whether in-house or outsourced, software development is still involved. An option which is increasingly taken – as in the case of the Brightmouth College payroll scenario – is to obtain a licence to run off-the-shelf software. The advantages of such an approach include:

- the supplier of the application can spread the cost of development over a large number of customers and thus the cost per customer should be reduced;
- the software already exists and so
  - it can be examined and perhaps even trialed before acquisition,
  - there is no delay while the software is being built;
- where lots of people have already used the software, most of the bugs are likely to have been reported and removed, leading to more reliable software.

However, there are disadvantages, which include the following.

- As you have the same application as everyone else, there is no competitive advantage.
- Modern off-the-shelf software tends to be very customizable: the characteristics of the application can be changed by means of various parameter tables. However, this flexibility has limits and you may end up having to change your office procedures in order to fit in with the computer system.
- You will not own the software code. This may rule out making modifications to the application in response to changes in the organization or its environment.
- Once you have acquired your off-the-shelf system, your organization may come to be very reliant upon it. This may create a considerable barrier to moving to a different application. The supplier may be in a position to charge inflated licence fees because you are effectively a captive customer.

Scanned by CamScanner

We will explore these issues further in Chapter 10 on managing contracts. In the remainder of this chapter we focus on situations where new software is being developed, whether in-house or outsourced.

4.3 Choosing Methodologies and Technologies

In the context of ICT system development and software engineering, the term *methodology* describes a collection of *methods*. We introduced 'method' in Chapter 1 as a general way of carrying out a specific task that could be applicable to any project needing to do that task. *Techniques* and methods are sometimes distinguished. Techniques tend to involve the application of scientific, mathematical or logical principles to resolve a particular kind of problem. They often require the practice of particular personal skills (the word 'technique' is derived from the Greek for skilful) – software design is a good example. Methods often involve the creation of *models*. A model is a representation of a system which abstracts certain features but ignores others. For example, an entity relationship diagram (ERD) is a model of the structure of the data used by a system. What can be confusing is that a software development life cycle itself is a type of system. Features of life cycles can therefore be abstracted and represented as 'models' as we will see later in this chapter. Some of these models can thus start to look a bit like methods.

Strictly speaking, methodology should refer to the 'study of methods'.

Project analysis should select the most appropriate methodologies and technologies for a project. Methodologies include approaches like the Unified Software Development Process (USDP), Structured Systems Analysis and Design Method (SSADM) and Human-Centred Design, while technologies might include appropriate application-building and automated testing environments. The analysis identifies the methodology, but also selects the methods within the methodology that are to be deployed.

As well as the products and activities, the chosen methods and technologies will affect:

- the training requirements for development staff;
- the types of staff to be recruited;
- the development environment – both hardware and software;
- system maintenance arrangements.

We are now going to describe some of the steps of project analysis.

Identify project as either objective-driven or product-driven

In Chapter 1 we distinguished between *objective-driven* and *product-driven* projects. A product-driven project creates products defined before the start of the project. An objective-driven project will often have come first which will have defined the general software solution that is to be implemented.

The soft systems approach is described in P. Checkland and J. Scholes (1999) *Soft Systems Methodology in Action*, John Wiley and Sons.

The project manager's dream is to have well-defined objectives but as much freedom as possible about how those objectives are to be satisfied. An objective might be to pay staff in a start-up company reliably, accurately and with low administrative costs. The company does not have to specify the use of a particular packaged software solution at the outset – but as we will see, there can be exceptions to this.

Sometimes the objectives of the project are uncertain or are the subject of disagreement. People might be experiencing problems but no one knows exactly how to solve these problems. ICT specialists might provide

Scanned by CamScanner

help with some problems but assistance from other specialisms might be needed with others. In these kinds of situation a *soft systems* approach might be considered.

Analyse other project characteristics

The following questions can be usefully asked.

We first introduced the difference between information systems and embedded systems in Chapter 1.

- *Is a data-oriented or process-oriented system to be implemented? Data-oriented* systems generally mean information systems that will have a substantial database. *Process-oriented* systems refer to embedded control systems. It is not uncommon to have systems with elements of both. Some writers suggest that the OO approach is more suitable for process-oriented systems where control is important than for systems dominated by a relational database.
- *Will the software that is to be produced be a general tool or application specific?* An example of a general tool would be a spreadsheet or a word processing package. An application-specific package could be, for example, an airline seat reservation system.
- *Are there specific tools available for implementing the particular type of application?* For example:

Note that here we are talking about writing the software tool, not its use.

  - *does it involve concurrent processing?* – the use of techniques appropriate to the analysis and design of such systems would be considered;
  - *will the system to be created be knowledge-based?* – expert systems have rules which result in some 'expert advice' when applied to a problem, and specific methods and tools exist for developing such systems; or
  - *will the system to be produced make heavy use of computer graphics?*
- *Is the system to be created safety critical?* For instance, could a malfunction in the system endanger human life? If so, among other things, testing would become very important.
- *Is the system designed primarily to carry out predefined services or to be engaging and entertaining?* With software designed for entertainment, design and evaluation will need to be carried out differently from more conventional software products.
- *What is the nature of the hardware/software environment in which the system will operate?* The environment in which the final software will operate could be different from that in which it is to be developed. Embedded software might be developed on a large development machine which has lots of supporting software tools such as compilers, debuggers and static analysers, but then be downloaded to a small processor in the target configuration. A standalone desktop application needs a different approach to one for a mainframe or a client–server environment.

EXERCISE 4.1

How would you categorize each of the following systems according to the classification above?

(a) a payroll system;

(b) a system to control a bottling plant;

(c) a system which holds details of the plans of plant used by a water company to supply water to consumers;

(d) a software package to support project managers;

(e) a system used by lawyers to access case law relating to company taxation.

Scanned by CamScanner

Identify high-level project risks

At the beginning of a project, some managers might expect elaborate plans even though we are ignorant of many important factors affecting the project. For example, until we have analysed the users' requirements in detail we cannot estimate the effort needed to build a system to meet those requirements. The greater the uncertainties at the beginning, the greater the risk that the project will be unsuccessful. Once we recognize an area of uncertainty we can, however, take steps to reduce its uncertainty.

Chapter 2 has already touched on some aspects of risk which are developed further in Chapter 7.

One suggestion is that uncertainty can be associated with the *products*, *processes*, or *resources* of a project.

- *Product uncertainty* How well are the requirements understood? The users themselves could be uncertain about what a proposed information system is to do. The government, say, might introduce a new form of taxation but its detailed operation might not be known until case law has been built up. Some environments change so quickly that a seemingly precise and valid statement of requirements rapidly becomes out of date.
- *Process uncertainty* The project under consideration might be the first where an organization is using an approach like extreme programming (XP) or a new application-building tool. Any change in the way that the systems are developed introduces uncertainty.

Extreme programming will be discussed in Section 4.15.

- *Resource uncertainty* The main area of uncertainty here is likely to be the availability of staff of the right ability and experience. The larger the number of resources needed or the longer the duration of the project, the more inherently risky it will be.

Of course, some risk factors can increase both uncertainty and complexity.

Some factors – such as continually changing requirements – increase *uncertainty*, while others – for instance, software size – increase *complexity*. Different strategies are needed to deal with the two distinct types of risks.

EXERCISE 4.2

At IOE, Amanda has identified possible user resistance as a risk to the annual maintenance contracts project. Would you classify this as a product, process or resource risk? It may be that it does not fit into any of these categories and some other is needed.

Brigette at Brightmouth College has identified as a risk the possibility that no suitable payroll package would be available on the market. What other risks might be inherent in the Brightmouth College payroll project?

Take into account user requirements concerning implementation

Chapter 13 on software quality discusses BS EN ISO 9001.

We suggested earlier that staff planning a project should try to ensure that unnecessary constraints are not imposed on the way that a project's objectives are to be met. The example given was the specification of the exact payroll package to be deployed. Sometimes, such constraints are unavoidable. International conglomerates have found that imposing uniform applications and technologies throughout all their component parts can save time and money. Obtaining IT services for the whole organization from a single supplier can mean that large discounts can be negotiated.

Scanned by CamScanner

A client organization often lays down standards that have to be adopted by any contractor providing software for them. Sometimes organizations specify that suppliers of software have BS EN ISO 9001 : 2000 or TickIT accreditation. This will affect the way projects are conducted.

Select general life-cycle approach

SSADM as a named methodology is now rarely used, but many of methods within it are still in wide use – sometimes under the general name of business system development (BSD) techniques.

- *Control systems* A real-time system will need to be implemented using an appropriate methodology. Real-time systems that employ concurrent processing may have to use techniques such as Petri nets.
- *Information systems* Similarly, an information system will need a methodology, such as SSADM or Information Engineering, that matches that type of environment. SSADM would be especially appropriate where the project employs a large number of development staff whose work will need to be coordinated: the method lays down in detail the activities and products needed at each step. Team members would therefore know exactly what is expected.
- *Availability of users* Where the software is for the general market rather than application and user specific, then a methodology which assumes that identifiable users exist who can be quizzed about their needs would have to be thought about with caution. Some business systems development methods assume an existing clerical system which can be analysed to yield the logical features of a new, computer-based, system. In these cases a marketing specialist may act as a surrogate user.
- *Specialized techniques* For example, expert system shells and logic-based programming languages have been invented to expedite the development of *knowledge-based systems*. Similarly, a number of specialized techniques and standard components are available to assist in the development of *graphics-based systems*.
- *Hardware environment* The environment in which the system is to operate could put constraints on the way it is to be implemented. The need for a fast response time or restricted computer memory might mean that only low-level programming languages can be used.
- *Safety-critical systems* Where safety and reliability are essential, this might justify the additional expense of a formal specification using a notation such as OCL. Extremely critical systems could justify the cost of having independent teams develop parallel systems with the same functionality. The operational systems can then run concurrently with continuous cross-checking. This is known as *n-version programming*.

OCL stands for Object Constraint Language.

The implications of prototyping and the incremental approach are explored later in the chapter.

- *Imprecise requirements* Uncertainties or a *novel hardware/software platform* mean that a prototyping approach should be considered. If the environment in which the system is to be implemented is a rapidly changing one, then serious consideration would need to be given to *incremental delivery*. If the users have *uncertain objectives* in connection with the project, then a *soft systems* approach might be desirable.

Scanned by CamScanner

EXERCISE 4.3

What, in broad outline, would be the most suitable approach for each of the following?

(a) a system which calculates the amount of a drug that should be administered to a patient who has a particular complaint;

(b) a system to administer a student loans scheme;

(c) a system to control trains in the Channel Tunnel.

4.4 Software Processes and Process Models

A software product development process usually starts when a request for the product is received from the customer. For a generic product, the marketing department of the company is usually considered as the customer. This expression of need for the product is called product inception. From the inception stage, a product undergoes a series of transformations through a few identifiable stages until it is fully developed and released to the customer. After release, the product is used by the customer and during this time the product needs to be maintained for fixing bugs and enhancing functionalities. This stage is called the maintenance stage. When the product is no longer useful to the customer, it is retired. This set of identifiable stages through which a product transits from inception to retirement form the life cycle of the product. The software life cycle is also commonly referred to as Software Development Life Cycle (SDLC) and software process.

A life cycle model (also called a process model) of a software product is a graphical or textual representation of its life cycle. Additionally, a process model may describe the details of various types of activities carried out during the different phases and the documents produced.

4.5 Choice of Process Models

The word 'process' emphasizes the idea of a system *in action*. In order to achieve an outcome, the system will have to execute one or more activities: this is its process. This applies to the development of computer-based applications. A number of interrelated activities have to be undertaken to create a final product. These activities can be organized in different ways and we can call these *process models*.

The planner selects methods and specifies how they are to be applied. Not all parts of a methodology such as USDP or SSADM will be compulsory. Many student projects have the rather basic failing that at the planning stage they claim that, say, SSADM is to be used: in the event all that is produced are a few SSADM fragments such as a top-level data flow diagram and a preliminary logical data structure diagram. If this is all the particular project requires, it should be explicitly stated.

4.6 Structure versus Speed of Delivery

The principle behind structured methods is 'get it right first time'.

Although some 'object-oriented' specialists might object(!), we include the OO approach as a structured method – after all, we hope it is not unstructured. Structured methods consist of sets of steps and rules which, when applied, generate system products such as use case diagrams. Each of these products is carefully defined. Such methods are more time consuming and expensive than more intuitive approaches. The pay-off, it is hoped, is a less error prone and more maintainable final system. This balance of costs and benefits is more likely to be

Scanned by CamScanner

justified on a large project involving many developers and users. Because of the additional effort needed and their greater applicability to large and complex projects, these are often called *heavyweight* methods.

It might be thought that users would generally welcome the more professional approach that structured methods imply. However, customers for software are concerned with getting working applications delivered quickly and at less cost and often see structured methods as unnecessarily bureaucratic and slow. One response to this has been *rapid application development* (RAD) which puts the emphasis on quickly producing prototypes of the software for users to evaluate.

Joint Application Development by Jane Wood and Denise Silver, Wiley and Sons, 1995, is a useful introduction to JAD.

The RAD approach does not preclude the use of some elements of structured methods such as the drafting of logical data structure diagrams but also adopts tactics such as *joint application development* (JAD) workshops. In these workshops, developers and users work together intensively for, say, three to five days and identify and agree fully documented system requirements. Often these workshops are conducted away from the normal business and development environments in *clean rooms*, specially conference rooms free from outside interruption and suitably furnished with whiteboards and other aids to communication. Advocates of JAD believe that these hot-house conditions can speed up communication and negotiation that might otherwise take several weeks or months.

Use of JAD does not mean that the project is not structured. The definition of the scope of the project, the initial research involving the interviewing of key personnel and the creation of preliminary data and process models would need to planned and executed before the JAD sessions were organized. The results of JAD sessions could be implemented using quite conventional methods.

Another way of speeding up delivery is simply to deliver less. This can be done by breaking a large development into a series of small increments, each of which delivers a small amount of useful functionality quickly.

Two competing pressures can be seen. One is to get the job done as quickly and cheaply as possible, and the other is to make sure that the final product has a robust structure which will be able to meet evolving needs. Later in this chapter and in Chapter 12 we will discuss the increasingly important topic of agile methods which focuses on lightweight processes. There is, however, a contrasting approach which is the attempt to create *model-driven architectures* (MDA). System development using MDA involves creating a platform-independent model (PIM) which specifies system functionality using UML diagrams supplemented by additional information recorded in the Object Constraint Language (OCL). A PIM is the logical structure that should apply regardless of the software and hardware environment in which the system is to be implemented. This can be transformed into a platform-specific model (PSM) that takes account of a particular development and implementation environment. A PSM can then be transformed into executable code to implement a working system. The goal is that once a PIM had been created the creation of PSMs and executable code will be automated. At present, the automation of these transformation processes is still being developed.

4.7 The Waterfall Model

This is the 'classical' model of system development that is also known as the *one-shot* or *once-through* model. As can be seen from the example in Figure 4.2, there is a sequence of activities working from top to bottom. The diagram shows some arrows pointing upwards and backwards. This indicates that a later stage may reveal the need for some extra work at an earlier stage, but this should definitely be the exception rather than the rule. After all, the flow of a waterfall should be downwards, with the possibility of just a little splashing

Scanned by CamScanner

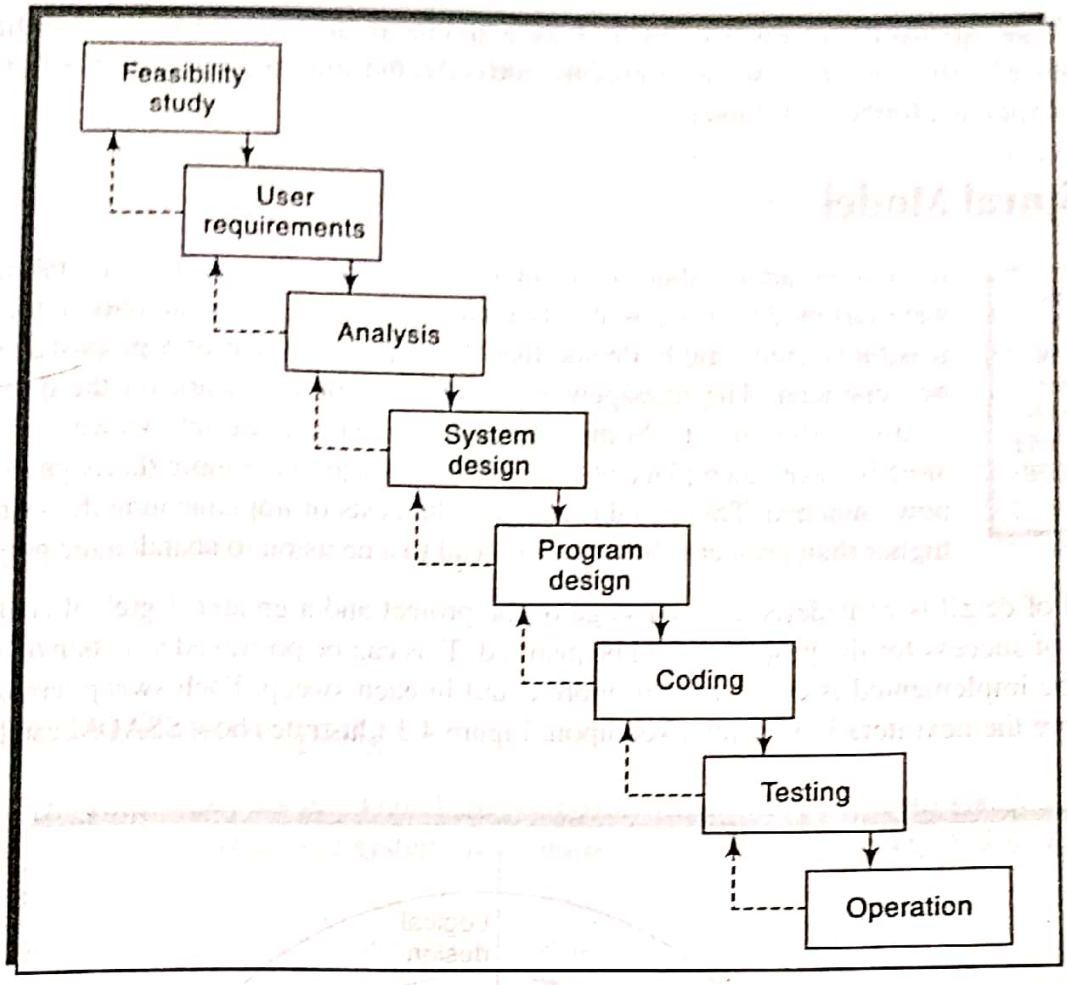

FIGURE 4.2 The waterfall model

back. The limited scope for iteration is in fact one of the strengths of this process model. With a large project you want to avoid reworking tasks previously thought to be completed. Having to reopen completed activities plays havoc with promised completion dates.

The first description of this approach is said to be that of H. D. Bennington in 'Production of Large Computer Programs' in 1956. This was reprinted in 1983 in *Annals of the History of Computing* 5(4).

The waterfall approach may be favoured by some managements because it creates natural milestones at the end of each phase. At these points, managers can review project progress to see whether the business case for the project is still valid. This is sometimes referred to as the stage-gate model. As we will see, stage-gates are compatible with process models other than the waterfall, but higher management may have to accept that activities may have to be grouped in different ways with these alternative approaches.

Even though writers often advocate alternative models, there is nothing intrinsically wrong with the waterfall approach in the right place. It is the ideal for which the project manager strives. Where the requirements are well defined and the development methods are well understood, the waterfall approach allows project completion times to be forecast with some confidence, allowing the effective control of the project. However, where there is uncertainty about how a system is to be implemented, and unfortunately there very often is, a more flexible, iterative, approach is required.

The waterfall model can expanded into the **V-process model** which is further explored in Section 13.11 on testing. This expansion is done by expanding the testing process into different types of testing which check

Scanned by CamScanner

the executable code against the products of each of the activities in the project life cycle leading up to the coding. For example, the code may seem to execute correctly, but may be at variance with the expected design. This is explained further in Chapter 13.

4.8 The Spiral Model

The original ideas behind the spiral model can be found in B. W. Boehm's 1988 paper 'A spiral model of software development and enhancement' in *IEEE Computer*, 21(5).

It could be argued that this is another way of looking at the waterfall model. In the waterfall model, it is possible to escape at the end of any activity in the sequence. A feasibility study might decide that the implementation of a proposed system would be beneficial. The management therefore authorize work on the detailed analysis of user requirements. Some analysis, for instance the interviewing of users, might already have taken place at the feasibility stage, but a more thorough investigation is now launched. This could reveal that the costs of implementing the system would be higher than projected benefits and lead to a decision to abandon the project.

A greater level of detail is considered at each stage of the project and a greater degree of confidence about the probability of success for the project should be justified. This can be portrayed as a loop or a spiral where the system to be implemented is considered in more detail in each sweep. Each sweep terminates with an evaluation before the next iteration is embarked upon. Figure 4.3 illustrates how SSADM can be interpreted in such a way.

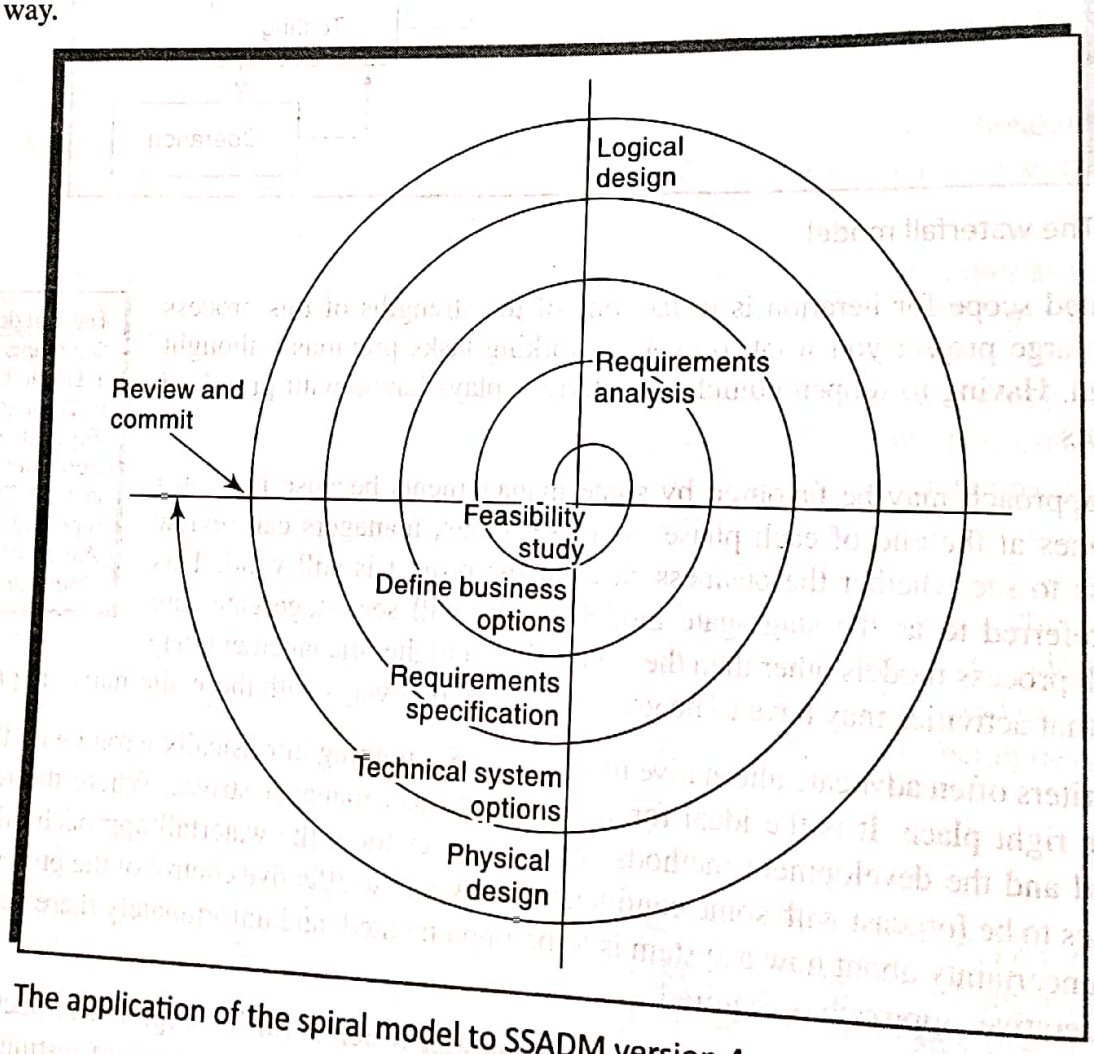

FIGURE 4.3 The application of the spiral model to SSADM version 4

Scanned by CamScanner

A key point here is that uncertainty about a project is usually because of a lack of knowledge about some aspect. We can spend money on activities at the start of the project that buy knowledge and reduce that uncertainty.

The distinguishing characteristic features of the spiral model are the incremental style of development and the ability to handle various types of risks. Each loop of the spiral is called a phase of this software process. In each phase, one or more features of the product are implemented after resolving any associated risks through prototyping. The exact number of loops of the spiral is not fixed and varies from one project to another. Note that the number of loops shown in Figure 4.4 is just an example illustrating how the spiral model can subsume SSADM. Each loop of the spiral is divided into four quadrants, indicating four stages in each phase. In the first stage of a phase, one or more features of the product are analysed and the risks in implementing those features are identified and resolved through prototyping. In the third stage, the identified features are implemented using the waterfall model. In the fourth and final stage, the developed increment is reviewed by the customer along with the development team and the features to be implemented next are identified. Note that the spiral model provides much more flexibility compared to the other models, in the sense that the exact number of phases through which the product is to be developed can be tailored by the project manager during execution of the project.

4.9 Software Prototyping

This is one way in which we can buy knowledge and reduce uncertainty. A prototype is a working model of one or more aspects of the projected system. It is constructed and tested quickly and inexpensively in order to test out assumptions.

Prototypes can be classified as throw-away or evolutionary.

- *Throw-away prototypes* The prototype tests out some ideas and is then discarded when the true development of the operational system is commenced. The prototype could be developed using a different software or hardware environment. For example, a desktop application builder could be used to evolve an acceptable user interface. A procedural programming language is then used for the final system where machine-efficiency is important.
- *Evolutionary prototypes* The prototype is developed and modified until it is finally in a state where it can become the operational system. In this case the standards that are used to develop the software have to be carefully considered.

Some of the reasons that have been put forward for prototyping follow.

- *Learning by doing* We can usually look back on a task and see where we have made mistakes.
- *Improved communication* Users do not get a feel for how the system is likely to work in practice from a specification.
- *Improved user involvement* The users can be more actively involved in design decisions.
- *Clarification of partially known requirements* Where there is no existing system to mimic, users can often get a better idea of what might be useful to them by trying out prototypes.
- *Demonstration of the consistency and completeness of a specification* Any mechanism that attempts to implement a specification on a computer is likely to uncover ambiguities and omissions. The humble spreadsheet can, for instance, check that calculations have been specified correctly.

The most important justification for a prototype is the need to reduce uncertainty by conducting an experiment.

Scanned by CamScanner

Some may argue, however, that this is a very dangerous suggestion.

- *Reduced need for documentation* Because a working prototype can be examined there is less need for detailed documentation of requirements.
- *Reduced maintenance costs* If the user is unable to suggest modifications at the prototyping stage they are more likely to ask for changes to the operational system. This reduction of maintenance costs is the core of the financial case for prototypes.
- *Feature constraint* If an application-building tool is used, then the prototype will tend to have features that are easily implemented by that tool. A paper-based design might suggest features that are expensive to implement.
- *Production of expected results* The problem with creating test cases is generally not the creation of the test input but the accurate calculation of the expected results. A prototype can help here.

Software prototyping is not without its drawbacks and dangers, however.

- *Users can misunderstand the role of the prototype* For example, they might expect the prototype to have as stringent input validation or as fast a response as the operational system, although this was not intended.
- *Lack of project standards possible* Evolutionary prototyping could just be an excuse for a sloppy 'hack it out and see what happens' approach.
- *Lack of control* It can be difficult to control the prototyping cycle if the driving force is the users' propensity to try out new things.
- *Additional expense* Building and exercising a prototype will incur additional expenses. However, this should not be over-estimated as many analysis and design tasks have to be undertaken whatever the approach.
- *Machine efficiency* A system built through prototyping, while sensitive to the users' needs, might not be as efficient in machine terms as one developed using more conventional methods.
- *Close proximity of developers* Prototyping could mean that code developers have to be sited close to the users. One trend is for organizations in developed countries to transfer software development to developing countries with lower costs such as India. Prototyping might prevent this.

4.10 Other Ways of Categorizing Prototypes

What is being learnt?

The most important reason for prototyping is a need to learn about an area of uncertainty. Thus it is essential to identify at the outset what is to be learnt from the prototype.

Computing students often realize that the software that they are to write as part of their final-year project could not safely be used by real users. They therefore call the software a 'prototype'. However, if it is a real prototype then they must:

- specify what they hope to learn from the prototype;
- plan how the prototype is to be evaluated;
- report on what has actually been learnt.

Prototypes can be used to find out about new development techniques, by using them in a pilot project. Alternatively, the development methods might be well known, but the nature of the application uncertain.

Scanned by CamScanner

Different projects will have uncertainties at different stages. Prototypes can therefore be used at different stages. A prototype might be used, for instance, at the requirements gathering stage to pin down requirements that seem blurred and shifting. A prototype might, on the other hand, be used at the design stage to test out the users' ability to navigate through a sequence of input screens.

To what extent is the prototyping to be done?

It would be unusual for the whole of the application to be prototyped. The prototyping usually simulates only some aspects of the target application. For example there might be:

- *Mock-ups* As when copies of input screens are shown to the users on a terminal, but the screens cannot actually be used.
- *Simulated interaction* For example, the user can type in a request to access a record and the system will show the details of a record, but the details shown are always the same and no access is made to a database.
- Partial working model:
  - *Vertical* Some, but not all, features are prototyped fully.
  - *Horizontal* All features are prototyped but not in detail – perhaps there is not full validation of input.

What is being prototyped?

- *The human–computer interface* With business applications, business process requirements have usually been established at an early stage. Prototyping tends, therefore, to be confined to the nature of operator interaction. Here the physical vehicle for the prototype should be as similar as possible to the operational system.
- *The functionality of the system* Here the precise way the system should function internally is not known. For example, a computer model of some real-world phenomenon is being developed. The algorithms used might need to be repeatedly adjusted until they satisfactorily imitate real-world behaviour.

EXERCISE 4.4

At what stage of a system development project (for example, feasibility study, requirements analysis, etc.) would a prototype be useful as a means of reducing the following uncertainties?

(a) There is a proposal that the senior managers of an insurance company have personal access to management information through an executive information system installed on personal computers located on their desks. Such a system would be costly to set up and there is some doubt about whether the managers would actually use the system.

(b) A computer system is to support sales office staff taking phone calls from members of the public enquiring about motor insurance and giving quotations over the phone.

(c) The insurance company is considering implementing the telephone sales system using the system development features supplied by Microsoft Access. They are not sure, at the moment, that it can provide the kind of interface that would be needed and are also concerned about the possible response times of a system developed using Microsoft Access.

Scanned by CamScanner

Controlling changes during prototyping

A major problem with prototyping is controlling changes to the prototype following suggestions by the users. One approach has been to categorize changes as belonging to one of three types:

- *Cosmetic* (often about 35% of changes)
  These are simple changes to the layout of the screens or reports. They are:
  (a) implemented;
  (b) recorded.
- *Local* (often about 60% of changes)
  These change the way that a screen or report is processed but do not affect other parts of the system. They are:
  (a) implemented;
  (b) recorded;
  (c) backed-up so that they can be removed at a later stage if necessary;
  (d) inspected retrospectively.
- *Global* (about 5% of changes)
  These are changes that affect more than one part of the processing. All changes here have to be the subject of a design review before they can be implemented.

Inspections are discussed in Chapter 13.

4.11 Incremental Delivery

This approach breaks the application down into small components which are then implemented and delivered in sequence. Each component delivered must give some benefit to the user. Figure 4.4 gives a general idea of the approach.

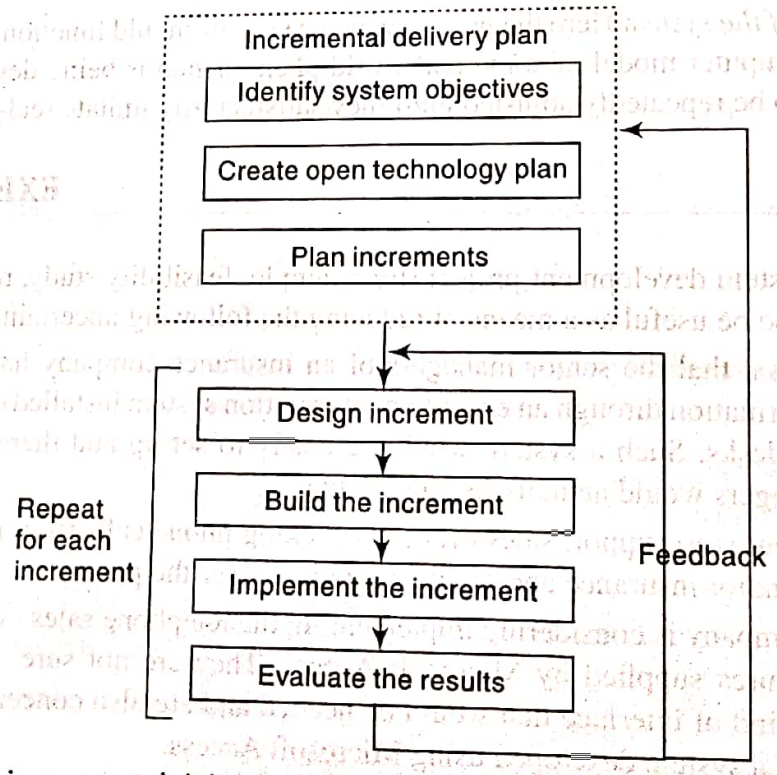

FIGURE 4.4 Intentional incremental delivery

Scanned by CamScanner

Time-boxing is often associated with an incremental approach. Here the scope of deliverables for an increment is rigidly constrained by an agreed deadline. This deadline has to be met, even at the expense of dropping some of the planned functionality. Omitted features can be transferred to later increments.

Advantages of this approach

These are some of the justifications given for the approach.

- The feedback from early increments improves the later stages.
- The possibility of changes in requirements is reduced because of the shorter time span between the design of a component and its delivery.
- Users get benefits earlier than with a conventional approach.
- Early delivery of some useful components improves cash flow, because you get some return on investment early on.
- Smaller sub-projects are easier to control and manage.
- *Gold-plating*, that is, the requesting of features that are unnecessary and not in fact used, is less as users know that if a feature is not in the current increment then it can be included in the next.
- The project can be temporarily abandoned if more urgent work emerges.
- Job satisfaction is increased for developers who see their labours bearing fruit at regular, short, intervals.

Tom Gilb, whose *Principles of Software Engineering Management* was published by Addison-Wesley in 1988, is a prominent advocate of this approach.

Disadvantages

On the other hand, these disadvantages have been put forward.

- Later increments might require modifications to earlier increments. This is known as *software breakage*.
- Software developers may be more productive working on one large system than on a series of smaller ones.
- Grady Booch, an authority on OO, suggests that with what he calls 'requirements driven' projects (which equate to incremental delivery) *'Conceptual integrity sometimes suffers because there is little motivation to deal with scalability, extensibility, portability or reusability beyond what any vague requirements might imply.'* Booch also suggests there could be a tendency towards a large number of discrete functions with little common infrastructure.

This quotation is from Grady Booch (1996) *Object Solutions: Managing the Object Oriented Project*, Addison-Wesley.

The incremental delivery plan

The nature and order of each increment to be delivered to the users have to be planned at the outset.

This process is similar to strategic planning but at a more detailed level. Attention is given to increments of a user application rather than whole applications. The elements of the incremental plan are the *system objectives, incremental plan* and the *open technology plan*.

The process of planning the increments of a project as described by Gilb has similarities with strategic planning described in Chapter 2.

Scanned by CamScanner

System objectives

Recall that earlier we suggested that project planners ideally want well-defined objectives, but as much freedom as possible about how these are to be met. These overall objectives can be expanded into more specific functional goals and quality goals.

Functional goals will include:

- objectives it is intended to achieve;
- jobs the system is to do;
- computer/non-computer functions to achieve them.

Chapter 13 discusses software quality characteristics.

In addition, measurable quality characteristics should be defined, such as reliability, response and security. If this is done properly these overarching quality requirements can go some way to meeting the concerns, expressed by Grady Booch, that these might get lost with the concentration on the requirements at increment level. It also reflects Tom Gilb's concern that system developers always keep sight of the objectives that they are trying to achieve on behalf of their clients. In the changing environment of an application individual requirements could change over the course of the project, but the objectives should not.

Open technology plan

If the system is to be able to cope with new components being continually added then it needs to be extendible, portable and maintainable.

As a minimum this will require the use of:

- a standard high-level language;
- a standard operating system;
- small modules;
- variable parameters, for example items such as the names of an organization and its departments, charge rates, and so on, are held in a parameter file that can be amended without programmer intervention;
- a standard database management system.

These are all things that might be expected as a matter of course in a modern software development environment.

Although Gilb does not suggest this, following Booch's hints it would be desirable to draw up an initial logical data model or object model for the whole system. It is difficult to see how the next stage of planning the scope and order of each increment could be done without this foundation.

Incremental plan

Having defined the overall objectives and an open technology plan, the next stage is to plan the increments using the following guidelines:

- Steps typically should consist of 1–5% of the total project.
- Non-computer steps should be included.
- An increment should, ideally, not exceed one month and should not, at worst, take more than three months.

Scanned by CamScanner

- Each increment should deliver some benefit to the user.
- Some increments will be physically dependent on others.
- In other cases value-to-cost ratios may be used to decide priorities (see below).

A non-computer step could be something like a streamlined clerical procedure.

A new system might be replacing an old computer system and the first increments could use parts of the old system. For example, the data for the database of the new system could initially be obtained from the old system's standing files.

Which steps should be first? Some steps will be prerequisites because of physical dependencies but others can be in any order. Value-to-cost ratios (see Table 4.1) can be used to establish the order in which increments are to be developed. The customer is asked to rate the value of each increment with a score in the range 1–10. The developers also rate the cost of developing each of the increments with a score in the range 0–10. This might seem rather crude, but people are often unwilling to be more precise. Dividing the value rating by the cost rating generates a ratio which indicates the relative 'value for money' of each increment.

TABLE 4.1 Ranking by value-to-cost ratio

Step	Value	Cost	Ratio	Rank
Profit reports	9	1	9	(2nd)
Online database	1	9	0.11	(6th)
Ad hoc enquiry	5	5	1	(4th)
Production sequence plans	2	8	0.25	(5th)
Purchasing profit factors	9	4	2.25	(3rd)
Clerical procedures	0	7	0	(7th)
Profit-based pay for managers	9	0	∞	(1st)

A zero cost would mean that the change can be implemented without software development – some costs might be incurred by users in changing procedures.

The value to cost ratio = V/C where V is a score 1–10 representing value to customer and C is a score 0–10 representing cost.

An incremental example

Tom Gilb describes a project where a software supplier negotiated a fixed-price contract with a three-month delivery time with the Swedish government to supply a system to support map-making. It later became apparent that the original estimate of effort upon which the bid was based was probably about half the real effort.

The project was replanned and divided into ten increments, each supplying something of use to the customer. The final increments were not available until three months after the contract's delivery date. The customer was not in fact unhappy about this as the most important parts of the application had actually been delivered early.

4.12 Atern/Dynamic Systems Development Method

In the United Kingdom, SSADM (Structured Systems Analysis and Design Method) has until recently been a predominant methodology. In no small part, this has been because of sponsorship by the United Kingdom government. More recently, however, it has lost some favour, partly because it has been perceived as overly

Scanned by CamScanner

bureaucratic and prescriptive. In contrast, there has been an increased interest in the iterative and incremental approaches we have outlined above. As a consequence, a consortium has developed guidelines for the use of such techniques and packaged the overall approach as the Dynamic Systems Development Method (DSDM). This has been re-badged as Atern. It is possible to attend courses on the method and to become an accredited Atern practitioner.

Eight core Atern principles have been enunciated.

1. *Focus on business need.* Every decision in the development process should be taken with a view to best satisfying business needs. Effectively this is emphasizing the need to avoid means–end inversion that we described in Section 4.1, that is, focusing on the detail of a procedure to the detriment of satisfactory project deliverables.
2. *Deliver on time.* Time-boxing is applied. Every deadline will see the delivery of valuable products, even if some less valuable ones are held over. This is better than delivery dates being pushed back until a delivery of all scheduled products can be made.
3. *Collaborate.* A one-team culture should be promoted, where user representatives are integrated into the delivery team.
4. *Never compromise quality.* Realistic quality targets are set early in the project. A process of continuously testing developing products starting as soon as possible is adopted.
5. *Develop iteratively.* The prototyping approach described in Section 4.8 would be an example of how this might be done.
6. *Build incrementally from firm foundations.* The incremental delivery approach as described in Section 4.11 is embraced.
7. *Communicate continuously.* In the case of users this could, for example, be done via workshops and the demonstration of prototypes.
8. *Demonstrate control.* Atern methodology has a range of plans and reports that can be used to communicate project intentions and outcomes to project sponsors and other management stakeholders.

JAD, Joint Application Development, was discussed in Section 4.6.

Figure 4.5 outlines the general approach. The main life cycle phases are shown:

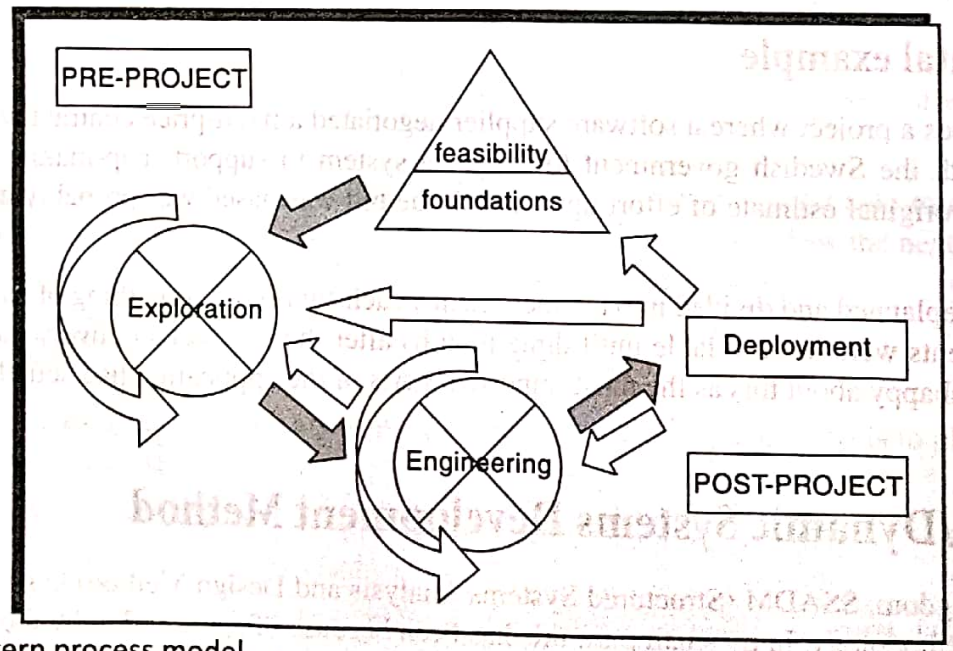

FIGURE 4.5 Atern process model

Scanned by CamScanner

P. BAGAVATHI SIVAKUMAR

- *Feasibility/foundation*. Among the activities undertaken here is derivation of a business case of the sort discussed in Chapter 2 and general outlines of the proposed architecture of the system to be developed.
- *Exploration cycle*. This investigates the business requirements. These requirements are translated into a viable design for the application. This could be an iterative process that could involve the creation of exploratory prototypes. A large project could be decomposed into smaller increments to assist the design process.
- *Engineering cycle*. This takes the design generated in the exploration cycle and converts it into usable components of the final system that will be used operationally. Once again this could be done using incremental and evolutionary techniques.
- *Deployment*. This gets the application created in the engineering cycle into actual operational use.

Not only can there be iterations within the exploration and engineering cycles, but an increment could involve requirements investigation followed by the building of the functionality.

Atern encourages the use of time-boxes. It is suggested that these should typically be between two and six weeks in order to make participants focus on real needs. It will be recalled that in order to meet the deadline imposed by a time-box, the implementation of less important features may be held over to later increments (or even dropped altogether). The relative importance of requirements can be categorized using the 'MoSCoW' classification:

Time-boxes were discussed in Section 4.11 on incremental delivery.

- *Must have*: that is, essential features.
- *Should have*: these would probably be mandatory if you were using a conventional development approach – but the system can operate without them.
- *Could have*: these requirements can be delayed with some inconvenience.
- *Won't have*: these features are wanted, but their delay to a later increment is readily accepted.

The possibility of requirements being reallocated to different increments means that project plans will need to be constantly updated if the project is to be successfully controlled.

4.13 Rapid Application Development

Rapid Application Development (RAD) model is also sometimes referred to as the rapid prototyping model. This model has the features of both the prototyping and the incremental delivery models.

The major aims of the RAD model are as follows:

- to decrease the time taken and the cost incurred to develop software systems; and
- to limit the costs of accommodating change requests by incorporating them as early as possible before large investments have been made on development and testing.

One of the major problems that has been identified with the waterfall model is the following. Clients often do not know what they exactly want until they see a working system. However, they do not see the working system until the development is complete in all respects and delivered to them. As a result, the exact requirements are brought out only through the process of customers commenting on the installed application. The required changes are incorporated through subsequent maintenance efforts. This makes the cost of accommodating any change extremely high. As a result, it takes a long time and enormous cost to have a good solution

Scanned by CamScanner

in place. Clearly, this model of developing software would displease even the best customer. The RAD model tries to overcome this problem by inviting and incorporating customer feedback on successively developed prototypes. In the RAD model, absence of long-term and detailed planning gives the flexibility to accommodate requirements change requests solicited from the customer during project execution.

In the RAD model, development takes place in a series of short cycles called iterations. Plans are made for one iteration at a time only. The time planned for each iteration is called a time box. Each iteration enhances the implemented functionality of the application a little. During each iteration, a quick and dirty prototype for some functionality is developed. The customer evaluates the prototype and gives feedback, based on which the prototype is refined. Thus, over successive iterations, the full set of functionalities of the software takes shape. However, it needs to be noted that in the RAD model, the prototype is used as an instrument for gathering customer feedback only and is not released to the customer for regular use.

The development team is also required to include a customer representative to clarify the requirements. Thus, conscious attempts are made to bridge the communication gap between the customer and the development team and to tune the system to the exact customer requirements. But, how does RAD model lead to faster product development? RAD emphasizes code reuse as an important means to get the work done fast. In fact, the RAD adopters were the earliest to embrace object-oriented languages and practices. RAD also advocates the use of specialized tools for faster creation of working prototypes.

4.14 Agile Methods

Agile methods are designed to overcome the disadvantages we have noted in our discussions on the heavyweight implementation methodologies. One of the main disadvantages of the traditional heavyweight methodologies is the difficulty of efficiently accommodating change requests from customers during execution of the project. Note that the agile model is an umbrella term that refers to a group of development processes, and not any single model of software development. There are various agile approaches such as the following:

- Crystal Technologies
- Atern (formerly DSDM)
- Feature-driven Development
- Scrum
- Extreme Programming (XP)

The Agile Manifesto is available at http://www.agilealliance.org

See S. Nerur, R. Mahapatra and G. Mangalara (2005) 'Challenges of migrating to agile methodologies' *Communications of the ACM* 48(5) 73–8.

In the agile model, the feature requirements are decomposed into several small parts that can be incrementally developed. The agile model adopts an iterative approach. Each incremental part is developed over an iteration. Each iteration is intended to be small and easily manageable, and lasts for a couple of weeks only. At a time, only one increment is planned, developed, and then deployed at the customer's site. No long-term plans are made. The time taken to complete an iteration is called a time box. The implication of the term 'time box' is that the end date for an iteration does not change. The development team can, however, decide to reduce the delivered functionality during a time box, if necessary, but the delivery date is considered sacrosanct.

Besides the delivery of increments after each time box, a few other principles discussed below are central to the agile model.

Scanned by CamScanner

- Agile model emphasizes face-to-face communication over written documents. Team size is deliberately kept small (5–9 people) to help the team members effectively communicate with each other and collaborate. This makes the agile model well-suited to the development of small projects, though large projects can also be executed using the agile model. In a large project, it is likely that the collaborating teams might work at different locations. In this case, the different teams maintain daily contact through video conferencing, telephone, e-mail, etc.

See H. Merisalo-Rantanen, T. Tuure and M. Rossi (2005) 'Is extreme programming just old wine in new bottles?' *Journal of Database Management* 16(4) 41–61.

- An agile project usually includes a customer representative in the team. At the end of each iteration, the customer representative along with the stakeholders review the progress made, re-evaluate the requirements, and give suitable feedback to the development team.
- Agile development projects usually deploy pair programming. In this approach, two programmers work together at one work station. One types the code while the other reviews the code as it is typed. The two programmers switch their roles every hour or so. Several studies indicate that programmers working in pairs produce compact well-written programs and commit fewer errors as compared to programmers working alone.

EXERCISE 4.5

As can be seen, there is much in common between the RAD model and the agile model. Identify the important differences between the agile model and the RAD model.

4.15 Extreme Programming (XP)

The primary source of information on XP is Kent Beck's *Extreme programming explained: embrace change*, first published in 1999 and updated in 2004. The description here is based on the first edition, with some comments where the ideas have been developed further in the second.

See Kent Beck (with Cynthia Andreas), *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1st edition 1999, 2nd edition 2004. 'Extreme programming' is sometimes shown with a capital 'X' i.e. 'eXtreme Programming'.

The ideas were largely developed on the C3 payroll development project at Chrysler. The approach is called 'extreme programming' because, according to Beck, '*XP takes commonsense principles to extreme levels*'. Four core values are presented as the foundations of XP.

1. **Communication and feedback.** It is argued that the best method of communication is face-to-face communication. Also, the best way of communicating to the users the nature of the software under production is to provide them with frequent working increments. Formal documentation is avoided.
2. **Simplicity.** The simplest design that implements the users' requirements should always be adopted. Effort should not be spent trying to cater for future possible needs – which in any case might never actually materialize.
3. **Responsibility.** The developers are the ones who are ultimately responsible for the quality of the software – not, for example, some system testing or quality control group.
4. **Courage.** This is the courage to throw away work in which you have already invested a lot of effort, and to start with a fresh design if that is what is called for. It is also the courage to try out new ideas – after

Scanned by CamScanner

all, if they do not work out, they can always be scrapped. Beck argues that this attitude is more likely to lead to better solutions.

Among the *core practices* of XP are the following.

The planning exercise

In the second edition of his book, Beck favours iterations of one week on the grounds that people tend to work naturally in weekly cycles.

Previously, when we talked about 'increments' we meant components of the system that users could actually use. XP refers to these as *releases*. Within these releases code is developed in *iterations*, periods of one to four weeks' duration during which specific features of the software are created. Note that these are not usually 'iterations' in the sense that they are new, improved, versions of the same feature – although this is a possibility. The planning game is the process whereby the features to be incorporated in the next release are negotiated. Each of the features is documented in a short textual description called a *story* that is written on a card. A process similar to value-to-cost ratio analysis discussed earlier in Section 4.11 or Atern's MoSCoW rating is carried out in order to give priorities to the features. At the time of the next code release, any features that have not been completed will be held over – that is, time-boxing is employed.

Small releases

The time between releases of functionality to users should be as short as possible. Beck suggests that releases should ideally take a month or two. This is compatible with Tom Gilb's recommendation of a month as the ideal time for an increment, with a maximum of three months.

Metaphor

The system to be built will be software code that reflects things that exist and happen in the real world. A payroll application will calculate and record payments to employees. The terms used to describe the corresponding software elements should, as far as possible, reflect real-world terminology – at a very basic level this would mean using meaningful names for variables and procedures such as 'hourly_rate' and 'calculate_gross_pay'. Beck suggests that what he calls the use of metaphor can do the job that 'system architecture' does on conventional projects. In this context 'architecture' refers to the use of system models such as class and collaboration diagrams to describe the system. The astute reader might point out that the use of the term 'architecture' is itself a metaphor.

Simple design

This is the practical implementation of the value of simplicity that was described above.

Testing

Testing is done at the same time as coding. The test inputs and expected results should be scripted so that the testing can be done using automated testing tools. These test cases can then be accumulated so that they can be used for regression testing to ensure that later developments do not insert errors into existing working code. This idea can be extended so that the tests and expected results are actually created before the code is created. Working out what tests are needed to check that a function is correct can itself help to clarify

Scanned by CamScanner

requirements. Two types of testing are needed: unit testing which focuses on the code a developer has just written, and function testing which is user-oriented and checks the correctness of a particular feature and which may involve several code units.

Refactoring

A threat to the target of striving to have always the simplest design is that over time, as modifications are made to code, the structure tends to become more spaghetti-like. The answer to this is to have the courage to resist the temptation to make changes that affect as little of the code as possible and be prepared to rewrite whole sections of code if this will keep the code structured. The repository of past test cases – see the section immediately above – can be executed to ensure that the refactoring has not introduced bugs into the application.

Pair programming

All software code is written by pairs of developers, one actually doing the typing and the other observing, discussing and making comments and suggestions about what the other is doing. At intervals, the developers can swap roles. The ideal is that you are constantly changing partners so that you get to know about a wide range of features that are under development. It follows from this that office environments need to be designed carefully to allow this type of working, and that developers will generally need to keep the same office hours.

Helen Sharp of the Open University has studied XP in practice. One of her observations is that the social nature of the development process encourages a rhythm of group meetings, pair working and daily 'builds' when new code is integrated that helps to give the project momentum. Interestingly, this rhythm of activity and review acting as a heart-beat of the project has also been noted in a successful dispersed project.

Collective ownership

This is really the corollary of pair programming. The team as a whole takes collective responsibility for the code in the system. A unit of code does not 'belong' to just one programmer who is the only one who can modify it.

Continuous integration

This is another aspect of testing practice. As changes are made to software units, integrated tests are run regularly – at least once a day – to ensure that all the components work together correctly.

Forty-hour weeks

Chapter 11 discusses, among other issues, the question of stress. It points out that working excessive hours (in some cases 60 hours or more a week) can lead to ill-health and be generally counterproductive. The principle is that normally developers should not work more than 40 hours a week. It is realistic to accept that sometimes there is a need for overtime work to deal with a particular problem – but in this case overtime should not be worked for two weeks in a row. Interestingly, in some case studies of the application of XP, the 40-hour rule was the only one not adhered to.

Scanned by CamScanner

On-site customers

Fast and effective communication with the users is achieved by having a user domain expert on-site with the developers.

Coding standards

If code is genuinely to be shared, then there must be common, accepted, coding standards to support the understanding and ease of modification of the code.

Limitations of XP

The successful use of XP is based on certain conditions. If these do not exist, then its practice could be difficult. These conditions include the following.

- There must be easy access to users, or at least a customer representative who is a domain expert. This may be difficult where developers and users belong to different organizations.
- Development staff need to be physically located in the same office.
- As users find out about how the system will work only by being presented with working versions of the code, there may be communication problems if the application does not have a visual interface.
- For work to be sequenced into small iterations of work, it must be possible to break the system functionality into relatively small and self-contained components.
- Large, complex systems may initially need significant architectural effort. This might preclude the use of XP.

XP does also have some intrinsic potential problems – particularly with regard to its reliance on tacit expertise and knowledge as opposed to externalized knowledge in the shape of documentation.

- There is a reliance on high-quality developers which makes software development vulnerable if staff turnover is significant.
- Even where staff retention is good, once an application has been developed and implemented, the tacit, personal, knowledge of the system may decay. This might make it difficult, for example, for maintenance staff without documentation to identify which bits of the code to modify to implement a change in requirements.
- Having a repository of comprehensive and accurate test data and expected results may not be as helpful as might be expected if the rationale for particular test cases is not documented. For example, where a change is made to the code, how do you know which test cases need to be changed?
- Some software development environments have focused on encouraging code reuse as a means of improving software development productivity. Such a policy would seem to be incompatible with XP.

4.16 Scrum

In the Scrum model, projects are divided into small parts of work that can be incrementally developed and delivered over time boxes that are called sprints. The product therefore gets developed over a series of manageable chunks. Each sprint typically takes only a couple of weeks. At the end of each sprint, stakeholders and team members meet to assess the progress and the stakeholders suggest to the development team any changes and improvements they feel necessary.

Scanned by CamScanner

In the scrum model, the team members assume three fundamental roles, viz., product owner, scrum master, and team member. The product owner is responsible for communicating the customer's vision of the product to the development team. The scrum master acts as a liaison between the product owner and the team, and facilitates the development work.

4.17 Managing Iterative Processes

This discussion of agile methods might be confusing as it seems to turn many of our previous planning concepts on their head.

Approaches like XP correctly emphasize the importance of communication and of removing artificial barriers to development productivity. XP to many might seem to be simply a 'licence to hack'. However, a more detailed examination of the techniques of XP shows that many (such as pair programming and installation standards) are conscious techniques to counter the excesses of hacking and to ensure that good maintainable code is written.

Booch suggests that there are two levels of development: the macro process and the micro process. The macro process is closely related to the waterfall process model. At this level, a range of activities carried out by a variety of specialist groups has to be coordinated. We need to have some dates when we know that major activities will be finished so that we know when we will need to bring in staff to work on subsequent activities. Within this macro process there will be micro process activities which might involve iterative working. Systems testing has always been one. Figure 4.6 illustrates how a sequential macro process can be imposed

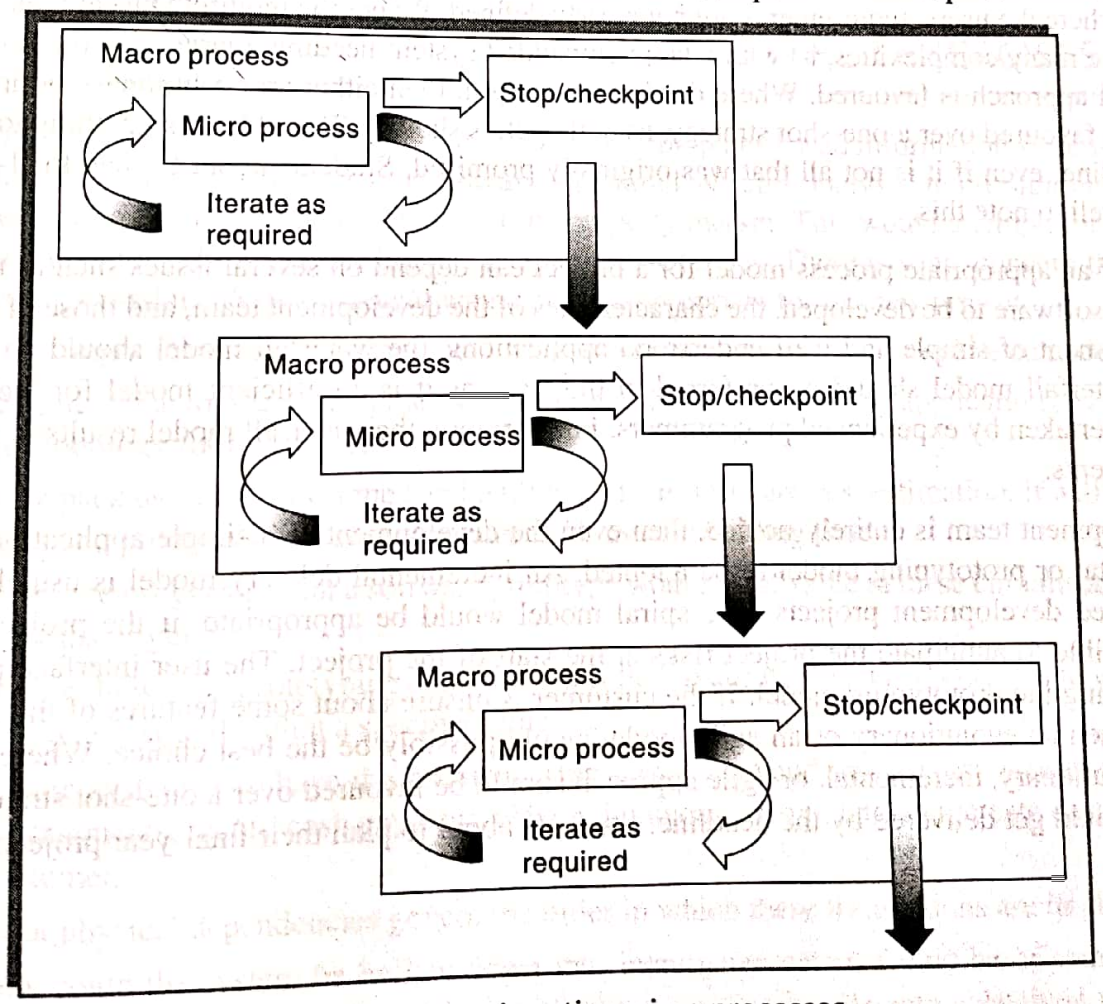

FIGURE 4.6 A macro process containing three iterative micro processes

Scanned by CamScanner

on a number of iterative sub-processes. With iterative micro processes, the use of time-boxes is needed to control at the macro level.

There are cases where the macro process itself can be iterative. It might be that a prototype for a complex technical system is produced in two or three successive versions, each taking several months to create and evaluate. In these circumstances, each iteration should be treated as a project in its own right.

D. Karlström and P. Runeson (2005). 'Combining agile methods with stage-gate project management' *IEEE Software*, May/June.

The packaging of micro processes within larger macro processes means that it is possible for agile projects using XP practices to exist within a more traditional stage-gate project environment (see Section 4.7) which has formal milestones where the business case for the project is reviewed. Agile projects might even contribute helpfully to this process as their progress is more visible.

4.18 Selecting the Most Appropriate Process Model

Construction of an application can be distinguished from its *installation*. It is possible to use different approaches for these two stages. For example, an application could be constructed using a waterfall or one-shot strategy but then be released to its users in increments. The only combinations of construction and installation strategies that are not feasible are the evolutionary installation with any construction approach other than evolutionary.

Whenever uncertainty is high, an evolutionary approach needs to be favoured. An example of this situation would be where the users' requirements are not clearly defined. Where the requirements are relatively certain but there are many complexities, as with a large embedded system needing a large amount of code, then an incremental approach is favoured. Where deadlines are tight, then either an evolutionary or an incremental approach is favoured over a one-shot strategy, as both tactics should allow at least something to be delivered at the deadline, even if it is not all that was originally promised. Students about to plan final-year projects would do well to note this.

Selection of an appropriate process model for a project can depend on several issues such as the characteristics of the software to be developed, the characteristics of the development team, and those of the customer. For development of simple and well-understood applications, the waterfall model should be sufficient. In fact, the waterfall model should be preferred in this case as it is an efficient model for well-understood projects undertaken by experienced programmers. Furthermore, the waterfall model results in production of good documents.

If the development team is entirely novice, then even the development of a simple application may require an incremental or prototyping model to be adopted. An incremental delivery model is usually suitable for object-oriented development projects. The spiral model would be appropriate, if the project is large and it is not possible to anticipate the project risks at the start of the project. The user interface part is usually developed using the prototyping model. If the customer is unsure about some features of the software to be developed, then an evolutionary or an agile model would possibly be the best choice. Where deadlines are tight, an evolutionary, incremental, or agile approach should be favoured over a one-shot strategy, as at least something would get delivered by the deadline. Students about to plan their final-year projects should make a note of this.

Scanned by CamScanner

EXERCISE 4.6

A travel agency needs software for automating its book-keeping activities. The set of activities to be automated are rather simple and are at present being carried out manually. The travel agency has indicated that it is unsure about the type of user interface which would be suitable for its employees and its customers. Would it be proper for a development team to use the spiral model for developing this software?

CONCLUSION

This chapter has stressed the need to examine each project carefully to see if it has characteristics which suggest a particular approach or process model. These characteristics might suggest the addition of specific activities to the project plan.

The classic waterfall process model, which attempts to minimize iteration, should lead to projects that are easy to control. Unfortunately, many projects do not lend themselves to this structure. Prototyping may be able to reduce project uncertainties by allowing knowledge to be bought through experimentation. The incremental approach encourages the execution of a series of small, manageable, 'mini-projects' but does have some costs.

FURTHER EXERCISES

1. A building society has a long history of implementing computer-based information systems to support the work of its branches. It uses a proprietary structured systems analysis and design method. It has been decided to create a computer model of the property market. This would attempt, for example, to calculate the effect of changes of interest rates on house values. There is some concern that the usual methodology used for IS development would not be appropriate for the new project.
   (a) Why might there be this concern and what alternative approaches should be considered?
   (b) Outline a plan for the development of the system which illustrates the application of your preferred methodology for this project.
2. A software package is to be designed and built to assist in software cost estimation. It will input certain parameters and produce initial cost estimates to be used at bidding time.
   (a) It has been suggested that a software prototype would be of value in these circumstances. Explain why this might be.
   (b) Discuss how such prototyping could be controlled to ensure that it is conducted in an orderly and effective way and within a specified time span.
3. An invoicing system is to have the following transactions: amend invoice, produce invoice, produce monthly statements, record cash payment, clear paid invoices from database, create customer records, delete customer.
   (a) What physical dependencies govern the order in which these transactions are implemented?
   (b) How could the system be broken down into increments which would be of some value to the users (**Hint** – think about the problems of taking existing details onto a database when a system is first implemented)?

Scanned by CamScanner

4. In Section 4.10 the need was stressed of defining what is to be learnt from a prototype and the way that it will be evaluated to obtain the new knowledge. Outline the learning outcomes and evaluation for the following.
   (a) A final-year degree student is to build an application that will act as a 'suggestions box' in a factory. The application will allow employees to make suggestions about process improvements, and will track the subsequent progress of the suggestion as it is evaluated. The student wants to use a web-based front-end with a conventional database. The student has not previously developed any applications using this mix of technologies.
   (b) An engineering company has to maintain a large number of different types of document relating to current and previous projects. It has decided to evaluate the use of a computer-based document retrieval system and wishes to try it out on a trial basis.
   (c) A business which specializes in 'e-solutions', that is, the development of business applications that exploit the World Wide Web, has been approached by the computing school of a local university. The school is investigating setting up a special website for its former students. The website's core will be information about job and training opportunities and it is hoped that this will generate income through advertising. It is agreed that some kind of pilot to evaluate the scheme is needed.
5. In a college environment, an intranet for students that holds information about courses, such as lecture programmes, reading lists and assignment briefs, is often set up. As a 'real' exercise, plan, organize and carry out a JAD session to design (or improve the design of) an intranet facility.

   This will require:

   - preliminary interviews with representative key stakeholders (for example, staff who might be supplying information for the intranet);
   - creation of documents for use in the JAD proceedings;
   - recording of the JAD proceedings;
   - creating a report which will present the findings of the JAD session.
6. What are the major shortcomings of the waterfall model? How have those shortcomings been overcome by the agile model?
7. Identify the pros and cons of using pair programming over programmers working alone. Based on your analysis, point out if there are any situations where the pair programming technique may not be suitable.

Scanned by CamScanner