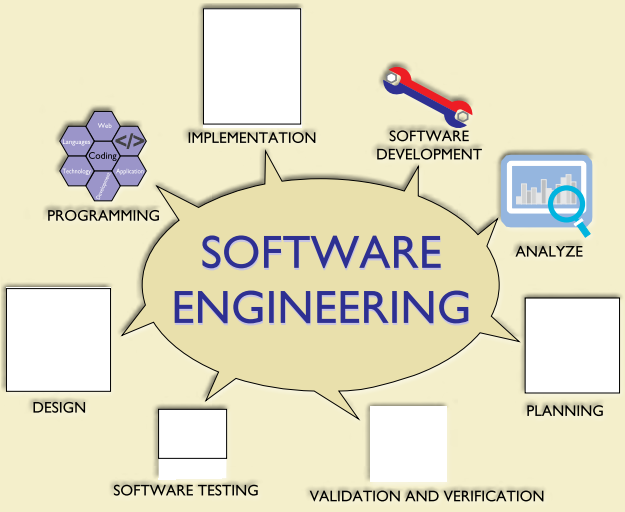
CHAPTER

2

SOFTWARE LIFE CYCLE MODELS



CHAPTER OUTLINE

- Classical waterfall model
- Relative effort distribution among different phases of development
- Iterative waterfall model
- V-model
- Prototyping model of software development
- Incremental software development
- Evolutionary model of software development
- Agile models: XP, Serum, and Lean
- Spiral model of software development Computer systems engineering

When to use iterative development? You should use iterative development only on projects that you want to succeed.

—Martin Fowler, in "UML Distilled"

LEARNING OBJECTIVES

- Basic concepts about software life cycle models
- Waterfall and related software development life cycle models
- Agile development models

In Chapter 1, we discussed a few basic issues in software engineering. We pointed out a few important differences between the exploratory program development style and the software engineering approach. Please recollect from our discussions in Chapter 1 that the exploratory style is also known as the *build* and *fix* programming. In build and fix programming, a programmer typically starts to write the program immediately after he has formed an informal understanding of the requirements. Once program writing is complete, he gets down to fix anything that would not meet the user's expectations. Usually, a large number of code fixes are required even for toy programs. This pushes up the development costs and pulls down the quality of the program. Further, this approach usually is a recipe for project failure when used to develop non-trivial programs requiring team effort. In contrast to the build and fix style, the software engineering approaches emphasise software development through a well-defined and ordered set of activities. These activities when graphically modelled (represented) and textually described are variously called as *software life cycle model*, *software development life cycle* (SDLC) *model*, and *software development process model*. Several life cycle models have so far been proposed. However, in this Chapter we confine our attention to only a few important and commonly used ones.

In this chapter, we first discuss a few basic concepts associated with life cycle models. Subsequently, we discuss the important activities that are required to be carried out in the classical waterfall model of software development. This is intended to provide an insight into the activities that are carried out as part of every life cycle model. In fact, the classical waterfall model can be considered as a basic model and all other life cycle models as extensions of this model to cater to specific project situations. After discussing the waterfall model, we discuss a few derivatives of this model. Subsequently we discuss the spiral model that generalises various life cycle models. Finally, we discuss a few recently proposed life cycle models that are categorised under the umbrella term *agile model*. Of late, agile models are finding increasing acceptance among developers and researchers.

The genesis of the agile model can be traced to the radical changes to the types of project that are being undertaken at present, rather than to any ground breaking innovations to the life cycle models themselves. The projects have changed from large multi-year product development projects couple of decades back to much smaller duration services projects now.

2.1 A FEW BASIC CONCEPTS

In this section, we present a few basic concepts concerning life cycle models.

Software life cycle

It is well known that all living organisms undergo a life cycle. For example when a seed is planted, it germinates, grows into a full tree, and finally dies. Based on this concept of a biological life cycle, the term *software life cycle* has been defined to imply the different stages (or phases) over which a software evolves from the initial customer request for it, to a fully developed software, and finally to a stage where it is no longer useful to any user, and is discarded.

As we have already pointed out, the life cycle of every software starts with a request expressing the need of the software by one or more customers. At this stage, the customers are usually not clear about all the features that would be needed, neither can they completely describe the identified features in concrete terms, and can only vaguely describe what is needed. This stage where the customer feels a need for the software and forms rough ideas about the required features is known as the *inception* stage. Starting with the inception stage, a software evolves through a series of identifiable stages (also called phases) on account of the development activities carried out by the developers, until it is fully developed and is released to the customers.

Once installed and made available for use, the users start to use the software. This signals the start of the operation (also called *maintenance*) phase. As the users use the software, not only do they request for fixing any failures that they might encounter, but they also continually suggest several improvements and modifications to the software. Thus, the maintenance phase usually involves continually making changes to the software to accommodate the bug-fix and change requests from the user. The operation phase is usually the longest of all phases and constitutes the useful life of a software. Finally the software is retired, when the users do not find it any longer useful due to reasons such as changed business scenario, availability of some new software having improved features and more efficient working, changed computing platforms, etc. This forms the essence of the life cycle of every software.

> The life cycle of a software represents the series of identifiable stages through which it evolves during its life time.

With this knowledge of a software life cycle, we discuss the concept of a software life cycle model and explore why it is necessary to follow a life cycle model in professional software development environments.

Software development life cycle (SDLC) model

In any systematic software development scenario, certain well-defined activities need to be performed by the development team and possibly by the customers as well, for the software to evolve from one stage in its life cycle to the next. For example, for a software to evolve from the requirements specification stage to the design stage, the developers need to elicit requirements from the customers, analyse those requirements, and formally document the requirements in the form of an SRS document. The customers need to review the SRS document, for the requirements phase to be complete.

A *software development life cycle* (SDLC) model (also called *software life cycle model* and *software development process model*) describes the different activities that need to be carried out for the software to evolve in its life cycle. Throughout our discussion, we shall use the terms *software development life cycle* (SDLC) and *software development process* interchangeably. However, some authors distinguish an SDLC from a software development process. In their usage, a software development process describes the life cycle activities more precisely and elaborately, as compared to an SDLC. Also, a development process may not only describe various activities that are carried out over the life cycle, but also prescribe a specific methodologies to carry out the activities, and also recommends the specific documents and other artifacts that should be produced at the end of each phase. In this sense, the term SDLC can be considered to be a more generic term, as compared to the development process and several development processes may fit the same SDLC.

An SDLC is represented graphically by drawing various stages of the life cycle and showing on it the transitions among the stages. This graphical model is usually accompanied by a textual description of various activities that need to be carried out during a phase before that phase can be considered to be complete.

An SDLC graphically depicts the different phases through which a software evolves. It is usually accompanied by a textual description of the different activities that need to be carried out during each phase.

Process *versus* methodology

Though the terms *process* and *methodology* are at time used interchangeably, there is a subtle difference between the two. First, the term process has a broader scope and addresses either all the activities taking place during software development, or certain coarse grained activities such as design (e.g. design process), testing (test process), etc. Further, a software process not only identifies the specific activities that need to be carried out, but may also prescribe certain methodology for carrying out each activity. For example, a design process may recommend that in the design stage, the high-level design activity be carried out using Hatley and Pirbhai's structured analysis and design methodology. A methodology, on the other hand, has a narrower scope and focuses on finer-grained activities as compared to a process, prescribes a set of steps for carrying out a specific life cycle activity. It may also include the rationale and philosophical assumptions behind the set of steps through which the activity is accomplished.

A software development process has a much broader scope as compared to a software development methodology. A process usually describes all the activities starting from the inception of a software to its maintenance and retirement stages, or at least a chunk of activities in the life cycle. It also recommends specific methodologies for carrying out each activity. A methodology, in contrast, describes the steps to carry out only a single or at best a few individual activities.

Why use a development process?

The primary advantage of using a development process is that it encourages development of software in a systematic and disciplined manner. Adhering to a process is especially important for development of professional software needing team effort. When software is

developed by a team rather than by an individual programmer, use of a life cycle model becomes indispensable for successful completion of the project.

Let us first consider the case of a single programmer developing a small program. An example of this is that a student is writing the program for a classroom assignment. The student might succeed even when he does not strictly follow a specific development process and adopts a build and fix style of development. However, it is a different ball game when a professional software is being developed by a team of programmers. Let us now understand the difficulties that may arise in this case if a team does not use any development process, and the team members are given complete freedom to develop their assigned part of the software as per their own discretion. Several types of problems may arise. We illustrate one of the problems using an example. Suppose, a software development problem has been divided into several parts and these parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part while making assumptions about the input results required from the other parts, another might decide to prepare the test documents first, and some other developer might start to carry out the design for the part assigned to him. In this case, severe problems can arise in interfacing the different parts and in managing the overall development. Therefore, *ad hoc* development turns out to be is a sure way to have a failed project. Believe it or not, this is exactly what has caused many projects to fail in the past!

Software development organisations have realised that adherence to a suitable life cycle model helps to produce good quality software and also helps to minimise the chances of time and cost overruns.

When a software is developed by a team, it is necessary to have a precise understanding among the team members as to—when to do what. In the absence of such an understanding, each member at any time would do whatever activity he feels like doing. This would be an open invitation to developmental chaos and project failure. The use of a suitable life cycle model is crucial to the successful completion of a team-based development project. But, do we need an SDLC model for developing a small program. In this context, we need to distinguish between programming-in-the-small and programming-in-the-large.

Programming-in-the-small refers to development of a toy program by a single programmer. On the other hand, *programming-in-the-large* refers to development of professional software through team effort. While development of software of the former type could succeed even when an individual programmer uses a build and fix style of development, use of a suitable SDLC is essential for a professional software development project involving team effort to succeed.

Why document a development process?

It is not enough for an organisation to just have a well-defined development process, but the development process needs to be properly documented. To understand the reason for this, let us consider that a development organisation does not document its development process. In this case, its developers only have an informal understanding of the development process. Informal understanding of the development process among the team members can create several problems during development. We have identified a few important problems

that may crop up when a development process is not adequately documented. Those problems are as follows:

- A documented process model ensures that every activity in the life cycle is accurately defined. Further, the methodologies for carrying out the activities are described. Without documentation, the activities and their ordering tend to be loosely defined, and different methodologies may be used by the developers even for the same activity. This can lead to confusion and misinterpretation. For example, code reviews may informally and inadequately be carried out by some team members if there is no documented methodology as to how code review should be done. Another difficulty is that for loosely defined activities, the developers tend to use their subjective judgments. As an example, unless it is explicitly prescribed, the team members would subjectively decide as to whether the test cases should be designed just after the requirements phase, after the design phase, or after the coding phase. Also, they would debate whether the test cases should be documented at all and the rigour with it should be documented.
- An undocumented process gives a clear indication to the members of the development teams about the lack of seriousness on the part of the management of the organisation about following the process. Therefore, an undocumented process serves as a hint to the developers to loosely follow the process. The symptoms of an undocumented process are easily visible—designs are shabbily done, reviews are not carried out rigorously, etc.
- A project team might often have to tailor a standard process model for use in a specific project. It is easier to tailor a documented process model, when it is required to modify certain activities or phases of the life cycle. For example, consider a project situation that requires some of the testing activities to be outsourced to another organisation. In this case, A documented process model would help to identify where exactly the required tailoring should occur.
- A documented process model, as we discuss later, is a mandatory requirement of the modern quality assurance standards such as ISO 9000 and SEI CMM. This means that unless a software organisation has a documented process, it would not qualify for accreditation by any of the quality certifying agencies. In the absence of a quality certification for the organisation, the customers would be suspicious of its capability of developing quality software and the organisation might find it difficult to win tenders for software development.

A documented development process forms a common understanding of the activities to be carried out among the software developers and helps them to develop software in a systematic and disciplined manner. A documented development process model, besides preventing the misinterpretations that might occur when the development process is not adequately documented, also helps to identify inconsistencies, redundancies, and omissions in the development process.

Nowadays, good software development organisations normally document their development process in the form of a booklet. They expect the developers recruited fresh to their organisation to first master their software development process during a short induction training that they are made to undergo.

Phase entry and exit criteria

A good SDLC besides clearly identifying the different phases in the life cycle, should unambiguously define the entry and exit criteria for each phase. The phase entry (or exit) criteria is usually expressed as a set of conditions that needs to be satisfied for the phase to start (or to complete). As an example, the phase exit criteria for the software requirements specification phase, can be that the *software requirements specification* (SRS) document is ready, has been reviewed internally, and also has been reviewed and approved by the customer. Only after these criteria are satisfied, the next phase can start.

If the entry and exit criteria for various phases are not well-defined, then that would leave enough scope for ambiguity in starting and ending various phases, and cause a lot of confusion among the developers. The decision regarding whether a phase is complete or not becomes subjective and it becomes difficult for the project manager to accurately tell how much has the development progressed. When the phase entry and exit criteria are not well-defined, the developers might close the activities of a phase much before they are actually complete, giving a false impression of rapid progress. In this case, it becomes very difficult for the project manager to determine the exact status of development and track the progress of the project. This usually leads to a problem that is usually identified as the *99 per cent complete syndrome*. This syndrome appears when there the software project manager has no definite way of assessing the progress of a project, the optimistic team members feel that their work is 99 per cent complete even when their work is far from completion—making all projections made by the project manager about the project completion time to be highly inaccurate.

2.2 WATERFALL MODEL AND ITS EXTENSIONS

The waterfall model and its derivatives were extremely popular in the 1970s and still are heavily being used across many development projects. The waterfall model is possibly the most obvious and intuitive way in which software can be developed through team effort. We can think of the waterfall model as a generic model that has been extended in many ways for catering to specific software development situations. This has yielded all other software life cycle models. For this reason, after discussing the classical and iterative waterfall models, we discuss its various extensions.

2.2.1 Classical Waterfall Model

Classical waterfall model is intuitively the most obvious way to develop software. It is simple but idealistic. In fact, it is hard to put this model into use in any non-trivial software development project, since developers do commit many mistakes during various development activities many of which are noticed only during a later phase. This requires to revisit the work of a previous phase to correct the mistakes, but the classical waterfall model has no provision to go back to modify the artifacts produced during an earlier phase. One might wonder if this model is hard to use in practical development projects, then why study it at all? The reason is that all other life cycle models can be thought of as being extensions of the classical waterfall model. Therefore, it makes sense to first understand the classical waterfall model, in order to be able to develop a proper understanding of other life cycle models. Besides, we shall see later in this text that this model though not used for software development; is implicitly used while documenting software.

The classical waterfall model divides the life cycle into six phases as shown in Figure 2.1. It can be easily observed from this figure that the diagrammatic representation of the classical waterfall model resembles a multi-level waterfall. This resemblance justifies the name of the model.

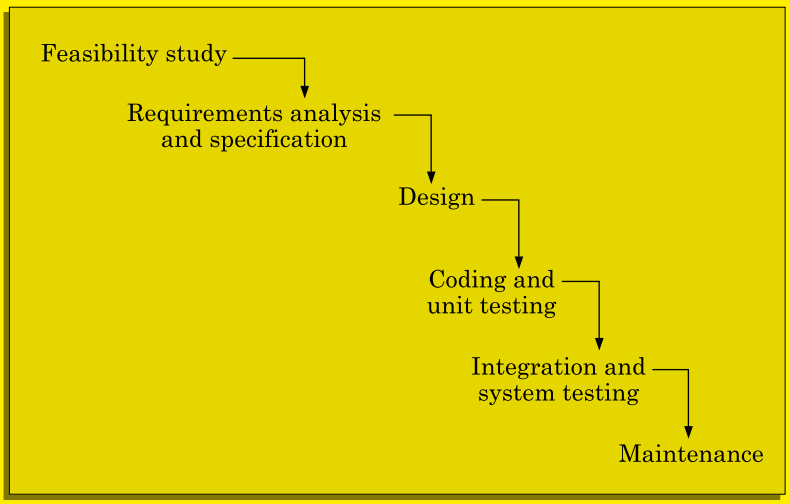


FIGURE 2.1 Classical waterfall model.

Phases of the classical waterfall model

The different phases of the classical waterfall model have been shown in Figure 2.1. As shown in Figure 2.1, the different phases are—feasibility study, requirements analysis and specification, design, coding and unit testing, integration and system testing, and maintenance. The phases starting from the feasibility study to the integration and system testing phase are known as the *development phases*. A software is developed during the development phases, and at the completion of the development phases, the software is delivered to the customer. After the delivery of software, customers start to use the software signalling the commencement of the operation phase. As the customers start to use the software, changes to it become necessary on account of bug fixes and feature extensions, causing maintenance works to be undertaken.

Therefore, the last phase is also known as the *maintenance phase* of the life cycle. It needs to be kept in mind that some of the text books have different number and names of the phases. An activity that spans all phases of software development is *project management*. Since it spans the entire project duration, no specific phase is named after it. Project management, nevertheless, is an important activity in the life cycle and deals with managing all the activities that take place during software development and maintenance.

In the waterfall model, different life cycle phases typically require relatively different amounts of efforts to be put in by the development team. The relative amounts of effort spent on different phases for a typical software has been shown in Figure 2.2. Observe from Figure 2.2 that among all the life cycle phases, the maintenance phase normally requires the maximum effort. On the average, about 60 per cent of the total effort put in by the development team in the entire life cycle is spent on the maintenance activities

alone. However, among the development phases, the integration and system testing phase requires the maximum effort in a typical development project. In the following subsection, we briefly describe the activities that are carried out in the different phases of the classical waterfall model.

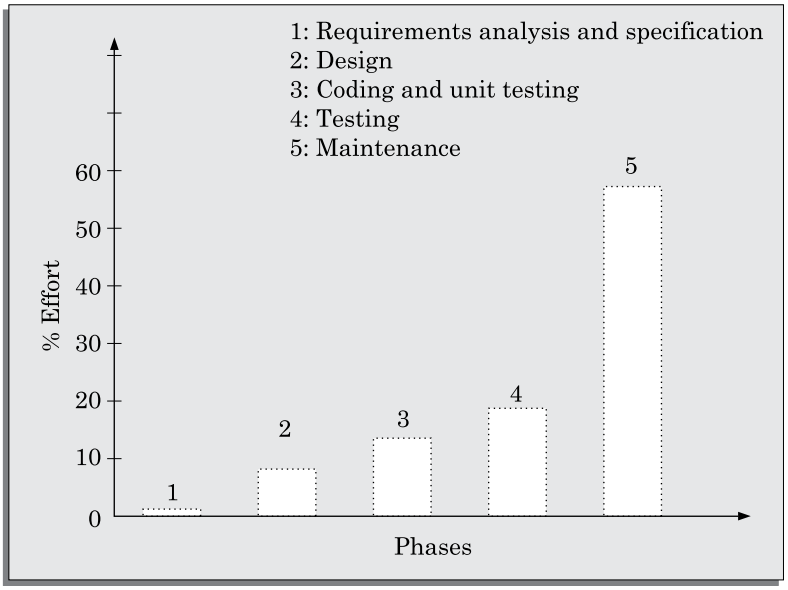


FIGURE 2.2 Relative effort distribution among different phases of a typical product.

Feasibility study

The main focus of the feasibility study stage is to determine whether it would be *financially* and *technically feasible* to develop the software. The feasibility study involves carrying out several activities such as collection of basic information relating to the software such as the different data items that would be input to the system, the processing required to be carried out on these data, the output data required to be produced by the system, as well as various constraints on the development. These collected data are analysed to perform the following:

Development of an overall understanding of the problem: It is necessary to first develop an overall understanding of what the customer requires to be developed. For this, only the important requirements of the customer need to be understood and the details of various requirements such as the screen layouts required in the *graphical user interface* (GUI), specific formulas or algorithms required for producing the required results, and the databases schema to be used are ignored.

Formulation of various possible strategies for solving the problem: In this activity, various possible high-level solution schemes to the problem are determined. For example, solution in a client-server framework and a standalone application framework may be explored.

Evaluation of the different solution strategies: The different identified solution schemes are analysed to evaluate their benefits and shortcomings. Such evaluation often requires making

approximate estimates of the resources required, cost of development, and development time required. The different solutions are compared based on the estimations that have been worked out. Once the best solution is identified, all activities in the later phases are carried out as per this solution. At this stage, it may also be determined that none of the solutions is feasible due to high cost, resource constraints, or some technical reasons. This scenario would, of course, require the project to be abandoned.

We can summarise the outcome of the feasibility study phase by noting that other than deciding whether to take up a project or not, at this stage very high-level decisions regarding the solution strategy is defined. Therefore, feasibility study is a very crucial stage in software development. The following is a case study of the feasibility study undertaken by an organisation. It is intended to give a feel of the activities and issues involved in the feasibility study phase of a typical software project.

CASE STUDY 2.1

A mining company named Galaxy Mining Company Ltd. (GMC Ltd.) has mines located at various places in India. It has about fifty different mine sites spread across eight states. The company employs a large number of miners at each mine site. Mining being a risky profession, the company intends to operate a special provident fund, which would exist in addition to the standard provident fund that the miners already enjoy. The main objective of having the *special provident fund* (SPF) would be to quickly distribute some compensation before the PF amount is paid.

According to this scheme, each mine site would deduct SPF installments from each miner every month and deposit the same to the central special provident fund commissioner (CSPFC). The CSPFC will maintain all details regarding the SPF installments collected from the miners.

GMC Ltd. requested a reputed software vendor Adventure Software Inc. to undertake the task of developing the software for automating the maintenance of SPF records of all employees. GMC Ltd. has realised that besides saving manpower on book-keeping work, the software would help in speedy settlement of claim cases. GMC Ltd. indicated that the amount it can at best afford ₹1 million for this software to be developed and installed.

Adventure Software Inc. deputed their project manager to carry out the feasibility study. The project manager discussed with the top managers of GMC Ltd. to get an overview of the project. He also discussed with the field PF officers at various mine sites to determine the exact details of the project. The project manager identified two broad approaches to solve the problem. One is to have a central database which would be accessed and updated via a satellite connection to various mine sites. The other approach is to have local databases at each mine site and to update the central database periodically through a dial-up connection. This periodic updates can be done on a daily or hourly basis depending on the delay acceptable to GMC Ltd. in invoking various functions of the software. He found that the second approach is very affordable and more fault-tolerant as the local mine sites can operate even when the communication link temporarily fails. In this approach, when a link fails, only the update of the central database gets delayed. Whereas in the first approach, all SPF work gets stalled at a mine site for the entire duration of link failure. The project manager quickly analysed the overall database functionalities required, the user-interface issues, and the software handling communication with the mine sites. From this analysis, he estimated the approximate cost to develop the software. He found that a solution involving maintaining local databases at the mine sites and periodically updating a central database is financially and technically feasible. The project manager discussed this solution with the president of GMC Ltd., who indicated that the proposed solution would be acceptable to them.

Requirements analysis and specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely requirements gathering and analysis, and requirements specification. In the following, we give an overview of these two activities:

- **Requirements gathering and analysis:** The goal of the requirements gathering activity is to collect all relevant information regarding the software to be developed from the customer with a view to clearly understand the requirements. For this, first requirements are gathered from the customer and then the gathered requirements are analysed. The goal of the requirements analysis activity is to weed out the incompleteness and inconsistencies in these gathered requirements. Note that an *inconsistent* requirement is one in which some part of the requirement contradicts some other part. On the other hand, an *incomplete* requirement is one in which some parts of the actual requirements have been omitted.
- **Requirements specification:** After the requirement gathering and analysis activities are complete, the identified requirements are documented. This document is called a *software requirements specification* (SRS) document. The SRS document is written using end-user terminology. This makes the SRS document understandable to the customer. Therefore, understandability of the SRS document is an important issue. The SRS document normally serves as a contract between the development team and the customer. Any future dispute between the customer and the developers can be settled by examining the SRS document. The SRS document is therefore an important document which must be thoroughly understood by the development team, and reviewed jointly with the customer. The SRS document not only forms the basis for carrying out all the development activities, but several documents such as users' manuals, system test plan, etc. are prepared directly based on it. In Chapter 4, we examine the requirements analysis activity and various issues involved in developing a good SRS document in more detail.

Design

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the *software architecture* is derived from the SRS document. Two distinctly different design approaches are popularly being used at present—the procedural and object-oriented design approaches. In the following, we briefly discuss the essence of these two approaches. These two approaches are discussed in detail in Chapters 6, 7, and 8.

- **Procedural design approach:** The traditional procedural design approach is in use in many software development projects at the present time. This traditional design technique is based on data flow modelling. It consists of two important activities; first *structured analysis* of the requirements specification is carried out where the data flow structure of the problem is examined and modelled. This is followed by a *structured design* step where the results of structured analysis are transformed into the software design.

During structured analysis, the functional requirements specified in the SRS document are decomposed into subfunctions and the data-flow among these subfunctions is analysed and represented diagrammatically in the form of DFDs. The DFD technique is discussed in Chapter 6. Structured design is undertaken once the structured analysis activity is complete. Structured design consists of two main activities—architectural design (also called *high-level design*) and detailed design (also called *Low-level design*). High-level design involves decomposing the system into modules, and representing the interfaces and the invocation relationships among the modules. A high-level software design is sometimes referred to as the *software architecture*. During the detailed design activity, internals of the individual modules such as the data structures and algorithms of the modules are designed and documented.

- **Object-oriented design approach:** In this technique, various objects that occur in the problem domain and the solution domain are first identified and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design. The OOD approach is credited to have several benefits such as lower development time and effort, and better maintainability of the software. The object-oriented design technique is discussed in Chapters 7 and 8.

Coding and unit testing

The purpose of the coding and unit testing phase is to translate a software design into source code and to ensure that individually each function is working correctly. The coding phase is also sometimes called the *implementation phase*, since the design is implemented into a workable solution in this phase. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually unit tested. The main objective of unit testing is to determine the correct working of the individual modules. The specific activities carried out during unit testing include designing test cases, testing, debugging to fix problems, and management of test cases. We shall discuss the coding and unit testing techniques in Chapter 10.

Integration and system testing

Integration of different modules is undertaken soon after they have been coded and unit tested. During the integration and system testing phase, the different modules are integrated in a planned manner. Various modules making up a software are almost never integrated in one shot (can you guess the reason for this?). Integration of various modules are normally carried out incrementally over a number of steps. During each integration step, previously planned modules are added to the partially integrated system and the resultant system is tested. Finally, after all the modules have been successfully integrated and tested, the full working system is obtained. System testing is carried out on this fully working system.

Integration testing is carried out to verify that the interfaces among different units are working satisfactorily. On the other hand, the goal of system testing is to ensure that the developed system conforms to the requirements that have been laid out in the SRS document.

System testing usually consists of three different kinds of testing activities:

- **α-testing:** α testing is the system testing performed by the development team.
- **β-testing:** This is the system testing performed by a friendly set of customers.
- **Acceptance testing:** After the software has been delivered, the customer performs system testing to determine whether to accept the delivered software or to reject it.

We discuss various integration and system testing techniques in more detail in Chapter 10.

Maintenance

The total effort spent on maintenance of a typical software during its operation phase is usually far greater than that required for developing the software itself. Many studies carried out in the past confirm this and indicate that the ratio of relative effort of developing a typical software product and the total effort spent on its maintenance is roughly 40:60. Maintenance is required in the following three types of situations:

- **Corrective maintenance:** This type of maintenance is carried out to correct errors that were not discovered during the product development phase.
- **Perfective maintenance:** This type of maintenance is carried out to improve the performance of the system, or to enhance the functionalities of the system based on customer's requests.
- **Adaptive maintenance:** Adaptive maintenance is usually required for porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system.

Various maintenance activities have been discussed in more detail in Chapter 13.

Shortcomings of the classical waterfall model

The classical waterfall model is a very simple and intuitive model. However, it suffers from several shortcomings. Let us identify some of the important shortcomings of the classical waterfall model:

No feedback paths: In classical waterfall model, the evolution of a software from one phase to the next is analogous to a waterfall. Just as water in a waterfall after having flowed down cannot flow back, once a phase is complete, the activities carried out in it and any artifacts produced in this phase are considered to be final and are closed for any rework. This requires that all activities during a phase are flawlessly carried out.

Contrary to a fundamental assumption made by the classical waterfall model, in practical development environments, the developers do commit a large number of errors in almost every activity they carry out during various phases of the life cycle. After all, programmers are humans and as the old adage says *to err is human*. The cause for errors can be many—oversight, wrong interpretations, use of incorrect solution scheme, communication gap, etc.

> The classical waterfall model is idealistic in the sense that it assumes that no error is ever committed by the developers during any of the life cycle phases, and therefore, incorporates no mechanism for error correction.

These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till the coding or testing phase. Once a defect is detected at a later time, the developers need to redo some of the work done during that phase and also redo the work of later phases that are affected by the rework. Therefore, in any non-trivial software development project, it becomes nearly impossible to strictly follow the classical waterfall model of software development.

Difficult to accommodate change requests: This model assumes that all customer requirements can be completely and correctly defined at the beginning of the project. There is much emphasis on creating an unambiguous and complete set of requirements. But, it is hard to achieve this even in ideal project scenarios. The customers' requirements usually keep on changing with time. But, in this model it becomes difficult to accommodate any requirement change requests made by the customer after the requirements specification phase is complete, and this often becomes a source of customer discontent.

Inefficient error corrections: This model defers integration of code and testing tasks until it is very late when the problems are harder to resolve.

No overlapping of phases: This model recommends that the phases be carried out sequentially—new phase can start only after the previous one completes. However, it is rarely possible to adhere to this recommendation and it leads to a large number of team members to idle for extended periods. For example, for efficient utilisation of manpower, the testing team might need to design the system test cases immediately after requirements specification is complete. (We shall discuss in Chapter 10 that the system test cases are designed solely based on the SRS document.) In this case, the activities of the design and testing phases overlap. Consequently, it is safe to say that in a practical software development scenario, rather than having a precise point in time at which a phase transition occurs, the different phases need to overlap for cost and efficiency reasons.

Is the classical waterfall model useful at all?

We have already pointed out that it is hard to use the classical waterfall model in real projects. In any practical development environment, as the software takes shape, several iterations through the different waterfall stages become necessary for correction of errors committed during various phases. Therefore, the classical waterfall model is hardly usable for software development. But, as suggested by Parnas [1972] the final documents for the product should be written as if the product was developed using a pure classical waterfall.

> Irrespective of the life cycle model that is actually followed for a product development, the final documents are always written to reflect a classical waterfall model of development, so that comprehension of the documents becomes easier for any one reading the document.

The rationale behind preparation of documents based on the classical waterfall model can be explained using Hoare's metaphor of mathematical theorem [1994] proving—A mathematician presents a proof as a single chain of deductions, even though the proof might have come from a convoluted set of partial attempts, blind alleys and backtracks. Imagine how difficult it would be to understand, if a mathematician presents a proof by retaining all the backtracking, mistake corrections, and solution refinements he made while working out the proof.

2.2.2 Iterative Waterfall Model

We had pointed out in the previous section that in a practical software development project, the classical waterfall model is hard to use. We had branded the classical waterfall model as an idealistic model. In this context, the iterative waterfall model can be thought of as incorporating the necessary changes to the classical waterfall model to make it usable in practical software development projects.

The main change brought about by the iterative waterfall model to the classical waterfall model is in the form of providing feedback paths from every phase to its preceding phases.

The feedback paths introduced by the iterative waterfall model are shown in Figure 2.3. The feedback paths allow for correcting errors committed by a programmer during some phase, as and when these are detected in a later phase. For example, if during the testing phase a design error is identified, then the feedback path allows the design to be reworked and the changes to be reflected in the design documents and all other subsequent documents. Please notice that in Figure 2.3 there is no feedback path to the feasibility stage. This is because once a team accepts to take up a project, it does not give up the project easily due to legal and moral reasons.

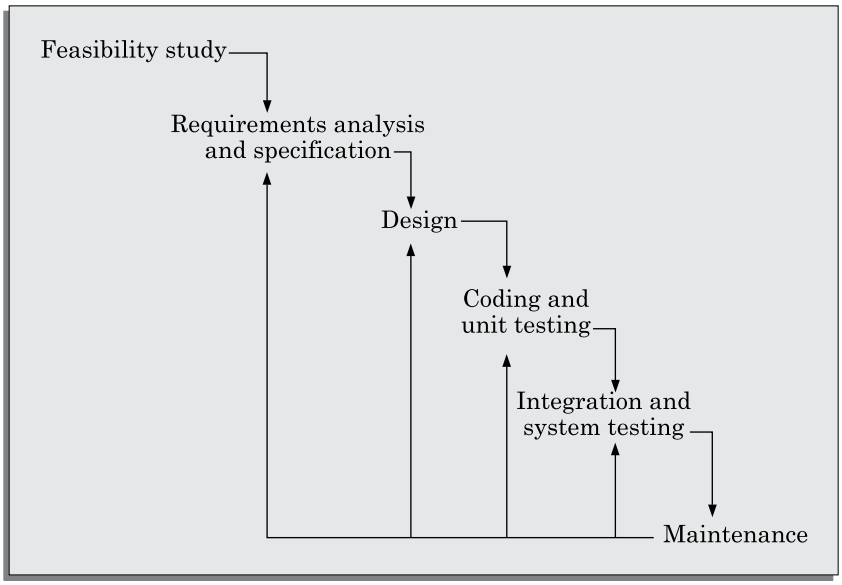


FIGURE 2.3 Iterative waterfall model.

Almost every life cycle model that we discuss are iterative in nature, except the classical waterfall model and the V-model—which are sequential in nature. In a sequential model, once a phase is complete, no work product of that phase are changed later.

Phase containment of errors

No matter how careful a programmer may be, he might end up committing some mistake or other while carrying out a life cycle activity. These mistakes result in errors (also called *faults* or *bugs*) in the work product. It is advantageous to detect these errors in the same

phase in which they take place, since early detection of bugs reduces the effort and time required for correcting those. For example, if a design problem is detected in the design phase itself, then the problem can be taken care of much more easily than if the error is identified, say, at the end of the testing phase. In the later case, it would be necessary not only to rework the design, but also to appropriately redo the relevant coding as well as the testing activities, thereby incurring higher cost. It may not always be possible to detect all the errors in the same phase in which they are made. Nevertheless, the errors should be detected as early as possible.

> The principle of detecting errors as close to their points of commitment as possible is known as *phase containment of errors*.

For achieving phase containment of errors, how can the developers detect almost all error that they commit in the same phase? After all, the end product of many phases are text or graphical documents, e.g. SRS document, design document, test plan document, etc. A popular technique is to rigorously review the documents produced at the end of a phase.

Phase overlap

Even though the strict waterfall model envisages sharp transitions to occur from one phase to the next (see Figure 2.3), in practice the activities of different phases overlap (as shown in Figure 2.4) due to two main reasons:

- In spite of the best effort to detect errors in the same phase in which they are committed, some errors escape detection and are detected in a later phase. These subsequently detected errors cause the activities of some already completed phases to be reworked. If we consider such rework after a phase is complete, we can say that the activities pertaining to a phase do not end at the completion of the phase, but overlap with other phases as shown in Figure 2.4.

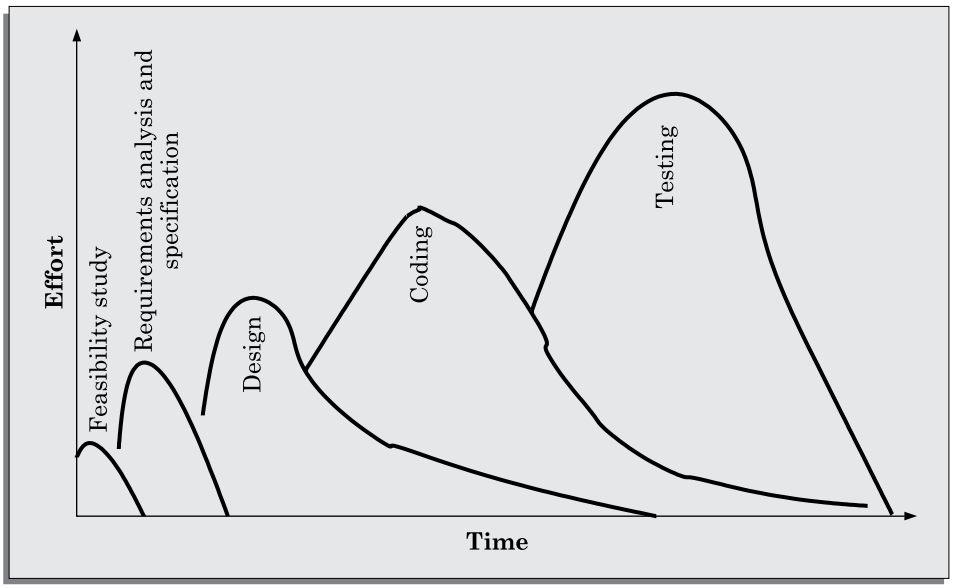


FIGURE 2.4 Distribution of effort for various phases in the iterative waterfall model.

- An important reason for phase overlap is that usually the work required to be carried out in a phase is divided among the team members. Some members may complete their part of the work earlier than other members. If strict phase transitions are maintained, then the team members who complete their work early would idle waiting for the phase to be complete, and are said to be in a *blocking state*. Thus the developers who complete early would idle while waiting for their team mates to complete their assigned work. Clearly this is a cause for wastage of resources and a source of cost escalation and inefficiency. As a result, in real projects, the phases are allowed to overlap. That is, once a developer completes his work assignment for a phase, proceeds to start the work for the next phase, without waiting for all his team members to complete their respective work allocations.

Considering these situations, the effort distribution for different phases with time would be as shown in Figure 2.4.

Shortcomings of the iterative waterfall model

The iterative waterfall model is a simple and intuitive software development model. It was used satisfactorily during 1970s and 1980s. However, the characteristics of software development projects have changed drastically over years. In the 1970s and 1960s, software development projects spanned several years and mostly involved generic software product development. The projects are now shorter, and involve Customised software development. Further, software was earlier developed from scratch. Now the emphasis is on as much reuse of code and other project artifacts as possible. Waterfall-based models have worked satisfactorily over last many years in the past. The situation has changed substantially now. As pointed out in the first chapter several decades back, every software was developed from scratch. Now, not only software has become very large and complex, very few (if at all any) software project is being developed from scratch. The software services (customised software) are poised to become the dominant types of projects. In the present software development projects, use of waterfall model causes several problems. In this context, the agile models have been proposed about a decade back that attempt to overcome the important shortcomings of the waterfall model by suggesting certain radical modification to the waterfall style of software development. In Section 2.4, we discuss the agile model. Some of the glaring shortcomings of the waterfall model when used in the present-day software development projects are as following:

Difficult to accommodate change requests: A major problem with the waterfall model is that the requirements need to be frozen before the development starts. Based on the frozen requirements, detailed plans are made for the activities to be carried out during the design, coding, and testing phases. Since activities are planned for the entire duration, substantial effort and resources are invested in the activities as developing the complete requirements specification, design for the complete functionality and so on. Therefore, accommodating even small change requests after the development activities are underway not only requires overhauling the plan, but also the artifacts that have already been developed.

Once requirements have been frozen, the waterfall model provides no scope for any modifications to the requirements.

While the waterfall model is inflexible to later changes to the requirements, evidence gathered from several projects points to the fact that later changes to requirements are almost inevitable. Even for projects with highly experienced professionals at all levels, as well as computer savvy customers, requirements are often missed as well as misinterpreted. Unless change requests are encouraged, the developed functionalities would be misfit to the true customer requirements. Requirement changes can arise due to a variety of reasons including the following—requirements were not clear to the customer, requirements were misunderstood, business process of the customer may have changed after the SRS document was signed off, etc. In fact, customers get clearer understanding of their requirements only after working on a fully developed and installed system.

> The basic assumption made in the iterative waterfall model that methodical requirements gathering and analysis alone is sufficient to comprehensively and correctly identify all the requirements by the end of the requirements phase is flawed.

Incremental delivery not supported: In the iterative waterfall model, the full software is completely developed and tested before it is delivered to the customer. There is no provision for any intermediate deliveries to occur. This is problematic because the complete application may take several months or years to be completed and delivered to the customer. By the time the software is delivered, installed, and becomes ready for use, the customer's business process might have changed substantially. This makes the developed application a poor fit to the customer's requirements.

Phase overlap not supported: For most real-life projects, it becomes difficult to follow the rigid phase sequence prescribed by the waterfall model. By the term *a rigid phase sequence*, we mean that a phase can start only after the previous phase is complete in all respects. As already discussed, strict adherence to the waterfall model creates *blocking states*. That is, if some team members complete their assigned work for a phase earlier than other team members, would have to ideal, and wait for the others to complete their work before initiating the work for the next phase. The waterfall model is usually adapted for use in real-life projects by allowing overlapping of various phases as shown in Figure 2.4.

Error correction unduly expensive: In waterfall model, validation is delayed till the complete development of the software. As a result, the defects that are noticed at the time of validation incur expensive rework and result in cost escalation and delayed delivery.

Limited customer interactions: This model supports very limited customer interactions. It is generally accepted that software developed in isolation from the customer is the cause of many problems. In fact, interactions occur only at the start of the project and at project completion. As a result, the developed software usually turns out to be a misfit to the customer's actual requirements.

Heavy weight: The waterfall model overemphasises documentation. A significant portion of the time of the developers is spent in preparing documents, and revising them as changes occur over the life cycle. Heavy documentation though useful during maintenance and for carrying out review, is a source of team inefficiency.

No support for risk handling and reuse: It becomes difficult to use the waterfall model in projects that are susceptible to various types of risks, or those involving significant reuse

of existing development artifacts. Please recollect that software services type of projects usually involve significant reuse of requirements, design and code. Therefore, it becomes difficult to use waterfall model in services type of projects.

2.2.3 V-Model

A popular development process model, V-model is a variant of the waterfall model. As is the case with the waterfall model, this model gets its name from its visual appearance (see Figure 2.5). In this model verification and validation activities are carried out throughout the development life cycle, and therefore the chances bugs in the work products considerably reduce. This model is therefore generally considered to be suitable for use in projects concerned with development of safety-critical software that are required to have high reliability.

As shown in Figure 2.5, there are two main phases—development and validation phases. The left half of the model comprises the development phases and the right half comprises the validation phases.

- In each development phase, along with the development of a work product, test case design and the plan for testing the work product are carried out, whereas the actual testing is carried out in the validation phase. This validation plan created during the development phases is carried out in the corresponding validation phase which have been shown by dotted arcs in Figure 2.5.

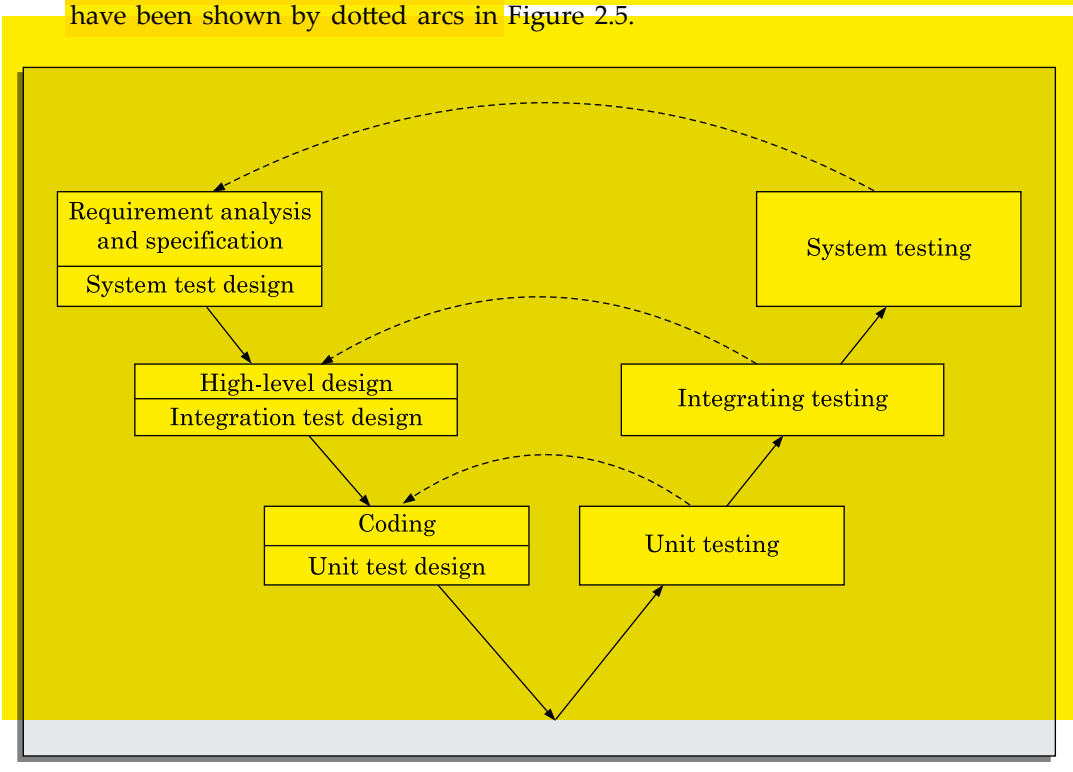


FIGURE 2.5 V-model.

- In the validation phase, testing is carried out in three steps—unit, integration, and system testing. The purpose of these three different steps of testing during the validation phase is to detect defects that arise in the corresponding phases of software development— requirements analysis and specification, design, and coding respectively.

V-model *versus* waterfall model

We have already pointed out that the V-model can be considered to be an extension of the waterfall model. However, there are major differences between the two. As already mentioned, in contrast to the iterative waterfall model where testing activities are confined to the testing phase only, in the V-model testing activities are spread over the entire life cycle. As shown in Figure 2.5, during the requirements specification phase, the system test suite design activity takes place. During the design phase, the integration test cases are designed. During coding, the unit test cases are designed. Thus, we can say that in this model, development and validation activities proceed hand in hand.

Advantages of V-model

The important advantages of the V-model over the iterative waterfall model are as following:

- In the V-model, much of the testing activities (test case design, test planning, etc.) are carried out in parallel with the development activities. Therefore, before testing phase starts significant part of the testing activities, including test case design and test planning, is already complete. Therefore, this model usually leads to a shorter testing phase and an overall faster product development as compared to the iterative model.
- Since test cases are designed when the schedule pressure has not built up, the quality of the test cases are usually better.
- The test team is reasonably kept occupied throughout the development cycle in contrast to the waterfall model where the testers are active only during the testing phase. This leads to more efficient manpower utilisation.
- In the V-model, the test team is associated with the project from the beginning. Therefore they build up a good understanding of the development artifacts, and this in turn, helps them to carry out effective testing of the software. In contrast, in the waterfall model often the test team comes on board late in the development cycle, since no testing activities are carried out before the start of the implementation and testing phase.

Disadvantages of V-model

Being a derivative of the classical waterfall model, this model inherits most of the weaknesses of the waterfall model.

2.2.4 Prototyping Model

The prototype model is also a popular life cycle model. The prototyping model can be considered to be an extension of the waterfall model. This model suggests building a working *prototype* of the system, before development of the actual software. A prototype is a toy and crude implementation of a system. It has limited functional capabilities, low reliability, or inefficient performance as compared to the actual software. A prototype can

be built very quickly by using several shortcuts. The shortcuts usually involve developing inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up rather than by performing the actual computations. Normally the term *rapid prototyping* is used when software tools are used for prototype construction. For example, tools based on *fourth generation languages* (4GL) may be used to construct the prototype for the GUI parts.

Necessity of the prototyping model

The prototyping model is advantageous to use for specific types of projects. In the following, we identify three types of projects for which the prototyping model can be followed to advantage:

- It is advantageous to use the prototyping model for development of the *graphical user interface* (GUI) part of an application. Through the use of a prototype, it becomes easier to illustrate the input data formats, messages, reports, and the interactive dialogs to the customer. This is a valuable mechanism for gaining better understanding of the customers' needs. In this regard, the prototype model turns out to be especially useful in developing the *graphical user interface* (GUI) part of a system. For the user, it becomes much easier to form an opinion regarding what would be more suitable by experimenting with a working user interface, rather than trying to imagine the working of a hypothetical user interface.

  The GUI part of a software system is almost always developed using the prototyping model.

- The prototyping model is especially useful when the exact technical solutions are unclear to the development team. A prototype can help them to critically examine the technical issues associated with product development. For example, consider a situation where the development team has to write a command language interpreter as part of a graphical user interface development. Suppose none of the team members has ever written a compiler before. Then, this lack of familiarity with a required development technology is a technical risk. This risk can be resolved by developing a prototype compiler for a very small language to understand the issues associated with writing a compiler for a command language. Once they feel confident in writing compiler for the small language, they can use this knowledge to develop the compiler for the command language. Often, major design decisions depend on issues such as the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype is often the best way to resolve the technical issues.

- An important reason for developing a prototype is that it is impossible to "get it right" the first time. As advocated by Brooks [1975], one must plan to throw away the prototype software in order to develop a good software later. Thus, the prototyping model can be deployed when development of highly optimised and efficient software is required.

  The prototyping model is considered to be useful for the development of not only the GUI parts of a software, but also for a software project for which certain technical issues are not clear to the development team.

Life cycle activities of prototyping model

The prototyping model of software development is graphically shown in Figure 2.6. As shown in Figure 2.6, software is developed through two major activities—prototype construction and iterative waterfall-based software development.

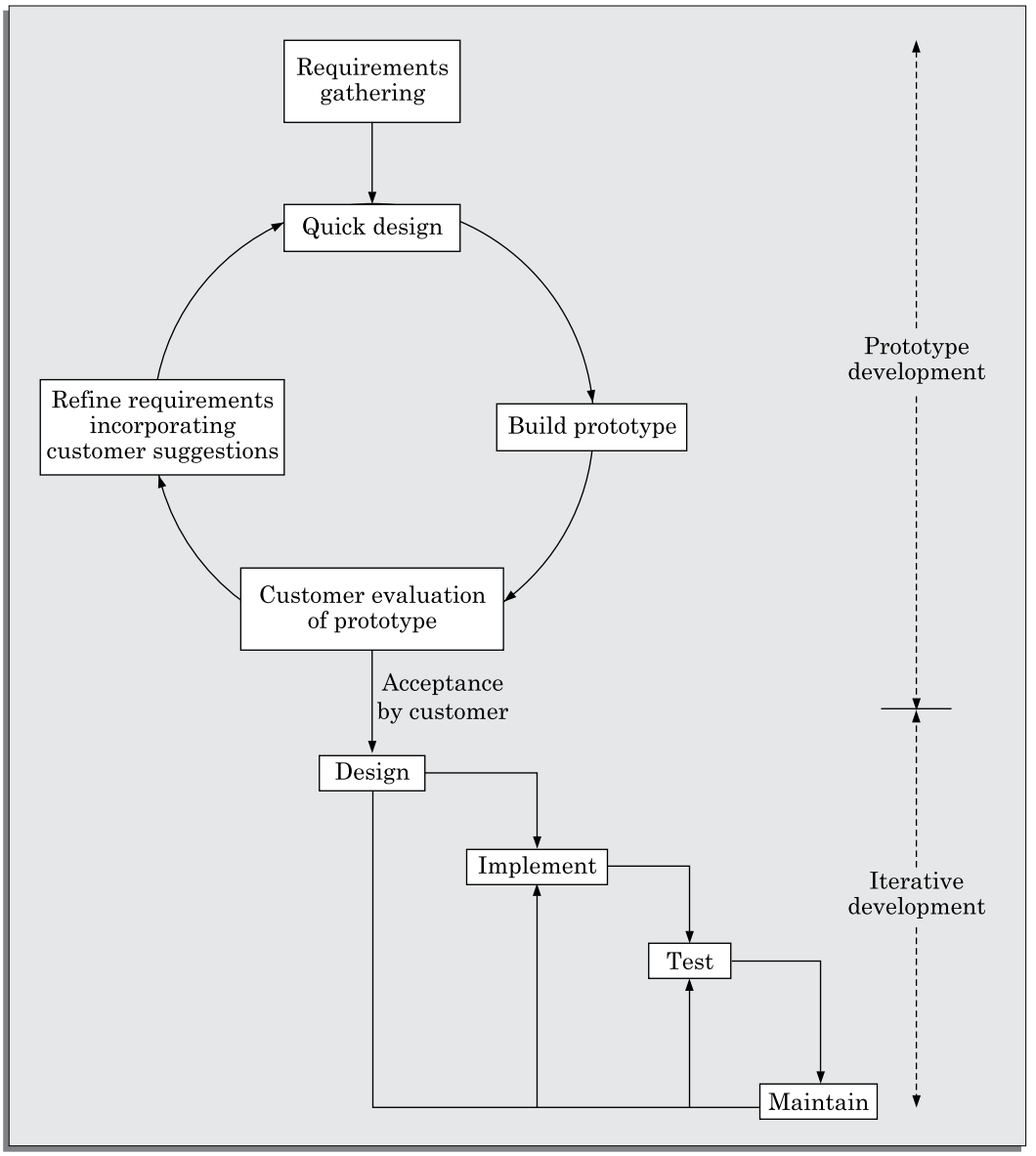


FIGURE 2.6 Prototyping model of software development.

Prototype development: Prototype development starts with an initial requirements gathering phase. A quick design is carried out and a prototype is built. The developed prototype is submitted to the customer for evaluation. Based on the customer feedback, the requirements are refined and the prototype is suitably modified. This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype.

Iterative development: Once the customer approves the prototype, the actual software is developed using the iterative waterfall approach. In spite of the availability of a working prototype, the SRS document is usually needed to be developed since the SRS document is invaluable for carrying out traceability analysis, verification, and test case design during later phases. However, for GUI parts, the requirements analysis and specification phase becomes redundant since the working prototype that has been approved by the customer serves as an animated requirements specification.

The code for the prototype is usually thrown away. However, the experience gathered from developing the prototype helps a great deal in developing the actual system.

By constructing the prototype and submitting it for user evaluation, many customer requirements get properly defined and technical issues get resolved by experimenting with the prototype. This minimises later change requests from the customer and the associated redesign costs.

Even though the construction of a throwaway prototype might involve incurring additional cost, for systems with unclear customer requirements and for systems with unresolved technical issues, the overall development cost usually turns out to be lower compared to an equivalent system developed using the iterative waterfall model.

Strengths of the prototyping model

This model is the most appropriate for projects that suffer from risks arising from technical uncertainties and unclear requirements. A constructed prototype helps overcome these risks.

Weaknesses of the prototyping model

The prototype model can increase the cost of development for projects that are routine development work and do not suffer from any significant risks. Even when a project is susceptible to risks, the prototyping model is effective only for those projects for which the risks can be identified upfront before the development starts. Since the prototype is constructed only at the start of the project, the prototyping model is ineffective for risks identified later during the development cycle. The prototyping model would not be appropriate for projects for which the risks can only be identified after the development is underway. For example, the risk of key personnel leaving the project midway is hard to predict at the start of the project.

2.2.5 Incremental Development Model

In the incremental life cycle model, the software is developed in increment. In this life cycle model, first the requirements are split into a set of increments. The first increment is a simple working system implementing only a few basic features. Over successive iterations, successive increments are implemented and delivered to the customer until the desired system is realised. The incremental development model has been shown in Figure 2.7.

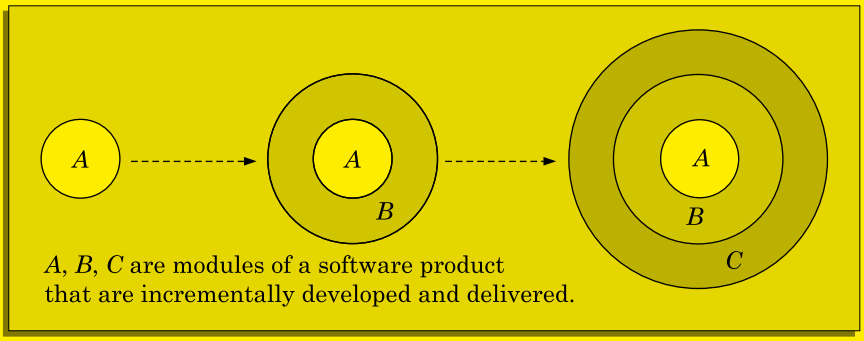


FIGURE 2.7 Incremental software development.

Life cycle activities of incremental development model

In the incremental life cycle model, the requirements of the software are first broken down into several modules or features that can be incrementally constructed and delivered. This has been pictorially depicted in Figure 2.7. At any time, plan is made only for the next increment and no long-term plans are made. Therefore, it becomes easier to accommodate change requests from the customers.

The development team first undertakes to develop the core features of the system. The core or basic features are those that do not need to invoke any services from the other features. On the other hand, non-core features need services from the core features. Once the initial core features are developed, these are refined into increasing levels of capability by adding new functionalities in successive versions. Each incremental version is usually developed using an iterative waterfall model of development. The incremental model is schematically shown in Figure 2.8. As each successive version of the software is constructed and delivered to the customer, the customer feedback is obtained on the delivered version and these feedbacks are incorporated in the next version. Each delivered version of the software incorporates additional features over the previous version and also refines the features that were already delivered to the customer.

The incremental model has schematically been shown in Figure 2.8. After the requirements gathering and specification, the requirements are split into several increments. Starting with the core (increment 1), in each successive iteration, the next increment is constructed using a waterfall model of development and deployed at the customer site. After the last (shown as increment n) has been developed and deployed at the client site, the full software is developed and deployed.

Advantages

The incremental development model offers several advantages. Two important ones are the following:

- **Error reduction:** The core modules are used by the customer from the beginning and therefore these get tested thoroughly. This reduces chances of errors in the core modules of the final product, leading to greater reliability of the software.

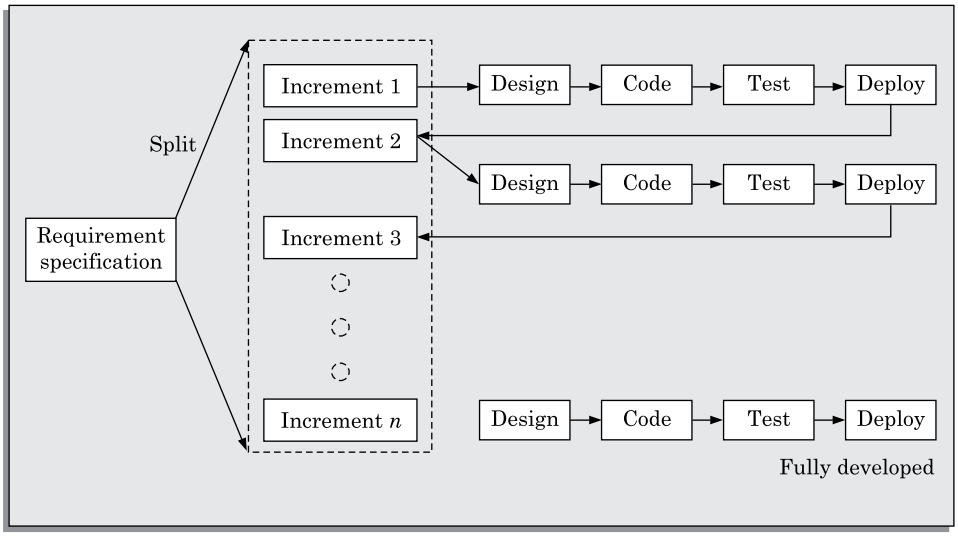


FIGURE 2.8 Incremental model of software development.

- **Incremental resource deployment:** This model obviates the need for the customer to commit large resources at one go for development of the system. It also saves the developing organisation from deploying large resources and manpower for a project in one go.

2.2.6 Evolutionary Model

This model has many of the features of the incremental model. As in case of the incremental model, the software is developed over a number of increments. At each increment, a concept (feature) is implemented and is deployed at the client site. The software is successively refined and feature-enriched until the full software is realised. The principal idea behind the evolutionary life cycle model is conveyed by its name. In the incremental development model, complete requirements are first developed and the SRS document prepared. In contrast, in the evolutionary model, the requirements, plan, estimates, and solution evolve over the iterations, rather than fully defined and frozen in a major up-front specification effort before the development iterations begin. Such evolution is consistent with the pattern of unpredictable feature discovery and feature changes that take place in new product development.

Though the evolutionary model can also be viewed as an extension of the waterfall model, but it incorporates a major paradigm shift that has been widely adopted in many recent life cycle models. Due to obvious reasons, the evolutionary software development process is sometimes referred to as *design a little, build a little, test a little, deploy a little* model. This means that after the requirements have been specified, the design, build, test, and deployment activities are iterated. A schematic representation of the evolutionary model of development has been shown in Figure 2.9.

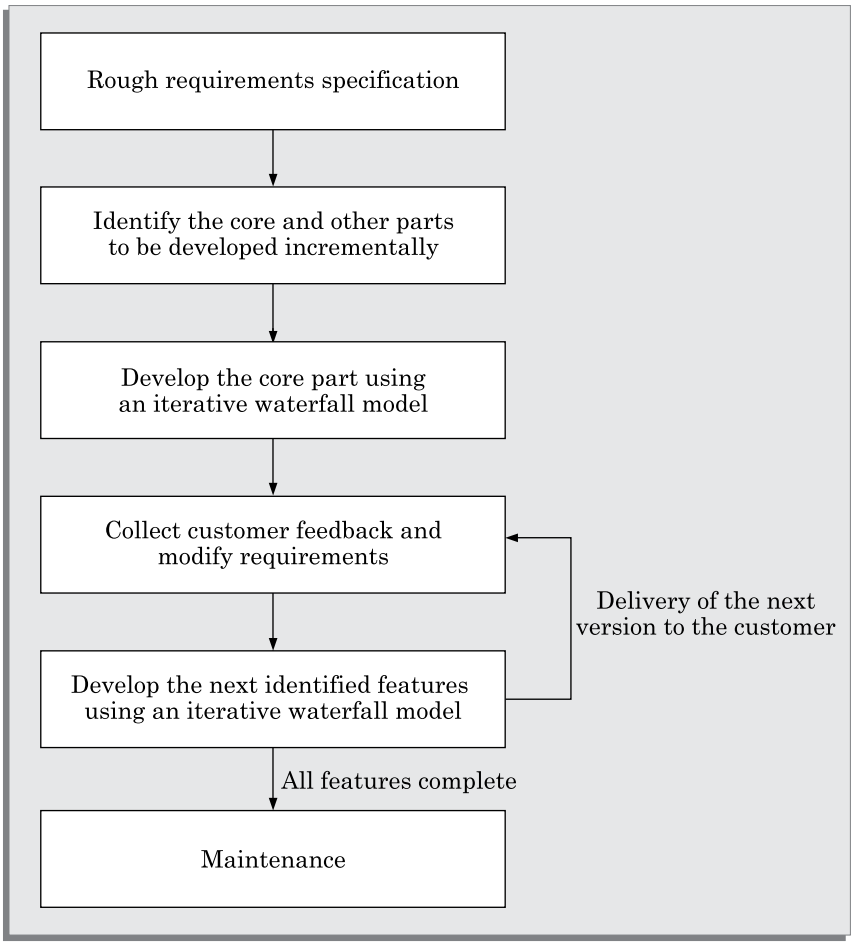


FIGURE 2.9 Evolutionary model of software development.

Advantages

The evolutionary model of development has several advantages. Two important advantages of using this model are the following:

- **Effective elicitation of actual customer requirements:** In this model, the user gets a chance to experiment with a partially developed software much before the complete requirements are developed. Therefore, the evolutionary model helps to accurately elicit user requirements with the help of feedback obtained on the delivery of different versions of the software. As a result, the change requests after delivery of the complete software gets substantially reduced.
- **Easy handling change requests:** In this model, handling change requests is easier as no long-term plans are made. Consequently, reworks required due to change requests are normally much smaller compared to the sequential models.

Disadvantages

The main disadvantages of the successive versions model are as follows:

- **Feature division into incremental parts can be non-trivial:** For many development projects, especially for small-sized projects, it is difficult to divide the required features into several parts that can be incrementally implemented and delivered. Further, even for larger problems, often the features are so intertwined and dependent on each other that even an expert would need considerable effort to plan the incremental deliveries.
- ***Ad hoc* design:** Since at a time design for only the current increment is done, the design can become ad hoc without specific attention being paid to maintainability and optimality. Obviously, for moderate sized problems and for those for which the customer requirements are clear, the iterative waterfall model can yield a better solution.

Evolutionary versus incremental model of developments

The evolutionary and incremental have several things in common, such as incremental development and deployment at the client site. However, in a purely incremental model, the requirement specification is completed before any development activities start. Once the requirement specification is completed, the requirements are split into requirements. In a purely evolutionary development, the development of the first version starts off after obtaining a rough understanding of what is required. As the development proceeds, more and more requirements emerge. The modern development models, such as the agile models are neither purely incremental, nor purely evolutionary, but are somewhat in between and are referred to as *incremental and evolutionary model*. In this mode, the initial requirements are obtained and specified, but requirements that emerge later are accommodated.

2.3 RAPID APPLICATION DEVELOPMENT (RAD)

The *rapid application development* (RAD) model was proposed in the early nineties in an attempt to overcome the rigidity of the waterfall model (and its derivatives) that makes it difficult to accommodate any change requests from the customer. It proposed a few radical extensions to the waterfall model. This model has the features of both prototyping and evolutionary models. It deploys an evolutionary delivery model to obtain and incorporate the customer feedbacks on incrementally delivered versions.

In this model prototypes are constructed, and incrementally the features are developed and delivered to the customer. But unlike the prototyping model, the prototypes are not thrown away but are enhanced and used in the software construction.

The major goals of the RAD model are as follows:

- To decrease the time taken and the cost incurred to develop software systems.
- To limit the costs of accommodating change requests.
- To reduce the communication gap between the customer and the developers.

Main motivation

In the iterative waterfall model, the customer requirements need to be gathered, analysed, documented, and signed off upfront, before any development could start. However, often

clients do not know what they exactly wanted until they saw a working system. It has now become well accepted among the practitioners that only through the process commenting on an installed application that the exact requirements can be brought out. But in the iterative waterfall model, the customers do not get to see the software, until the development is complete in all respects and the software has been delivered and installed. Naturally, the delivered software often does not meet the customer expectations and many change request are generated by the customer. The changes are incorporated through subsequent maintenance efforts. This made the cost of accommodating the changes extremely high and it usually took a long time to have a good solution in place that could reasonably meet the requirements of the customers. The RAD model tries to overcome this problem by inviting and incorporating customer feedback on successively developed and refined prototypes.

2.3.1 Working of RAD

In the RAD model, development takes place in a series of short cycles or iterations. At any time, the development team focuses on the present iteration only, and therefore plans are made for one increment at a time. The time planned for each iteration is called a *time box*. Each iteration is planned to enhance the implemented functionality of the application by only a small amount. During each time box, a quick-and-dirty prototype-style software for some functionality is developed. The customer evaluates the prototype and gives feedback on the specific improvements that may be necessary. The prototype is refined based on the customer feedback. Please note that the prototype is not meant to be released to the customer for regular use though.

The development team almost always includes a customer representative to clarify the requirements. This is intended to make the system tuned to the exact customer requirements and also to bridge the communication gap between the customer and the development team. The development team usually consists of about five to six members, including a customer representative.

How does RAD facilitate accommodation of change requests?

The customers usually suggest changes to a specific feature only after they have used it. Since the features are delivered in small increments, the customers are able to give their change requests pertaining to a feature already delivered. Incorporation of such change requests just after the delivery of an incremental feature saves cost as this is carried out before large investments have been made in development and testing of a large number of features.

How does RAD facilitate faster development?

The decrease in development time and cost, and at the same time an increased flexibility to incorporate changes are achieved in the RAD model in two main ways—minimal use of planning and heavy reuse of any existing code through rapid prototyping. The lack of long-term and detailed planning gives the flexibility to accommodate later requirements changes. Reuse of existing code has been adopted as an important mechanism of reducing the development cost.

RAD model emphasises code reuse as an important means for completing a project faster. In fact, the adopters of the RAD model were the earliest to embrace object-oriented languages and practices. Further, RAD advocates use of specialised tools to facilitate fast

creation of working prototypes. These specialised tools usually support the following features:

- Visual style of development.
- Use of reusable components.

2.3.2 Applicability of RAD Model

The following are some of the characteristics of an application that indicate its suitability to RAD style of development:

- **Customised software:** As already pointed out a customised software is developed for one or two customers only by adapting an existing software. In customised software development projects, substantial reuse is usually made of code from pre-existing software. For example, a company might have developed a software for automating the data processing activities at one or more educational institutes. When any other institute requests for an automation package to be developed, typically only a few aspects needs to be tailored—since among different educational institutes, most of the data processing activities such as student registration, grading, fee collection, estate management, accounting, maintenance of staff service records, etc. are similar to a large extent. Projects involving such tailoring can be carried out speedily and cost-effectively using the RAD model.
- **Non-critical software:** The RAD model suggests that a quick and dirty software should first be developed and later this should be refined into the final software for delivery. Therefore, the developed product is usually far from being optimal in performance and reliability. In this regard, for well understood development projects and where the scope of reuse is rather restricted, the iterative waterfall model may provide a better solution.
- **Highly constrained project schedule:** RAD aims to reduce development time at the expense of good documentation, performance, and reliability. Naturally, for projects with very aggressive time schedules, RAD model should be preferred.
- **Large software:** Only for software supporting many features (large software) can incremental development and delivery be meaningfully carried out.

Application characteristics that render RAD unsuitable

The RAD style of development is not advisable if a development project has one or more of the following characteristics:

- **Generic products (wide distribution):** As we have already pointed out in Chapter 1, software products are generic in nature and usually have wide distribution. For such systems, optimal performance and reliability are imperative in a competitive market. As it has already been discussed, the RAD model of development may not yield systems having optimal performance and reliability.
- **Requirement of optimal performance and/or reliability:** For certain categories of products, optimal performance or reliability is required. Examples of such systems include an operating system (high reliability required) and a flight

simulator software (high performance required). If such systems are to be developed using the RAD model, the desired product performance and reliability may not be realised.

- **Lack of similar products:** If a company has not developed similar software, then it would hardly be able to reuse much of the existing artifacts. In the absence of sufficient plug-in components, it becomes difficult to develop rapid prototypes through reuse, and use of RAD model becomes meaningless.
- **Monolithic entity:** For certain software, especially small-sized software, it may be hard to divide the required features into parts that can be incrementally developed and delivered. In this case, it becomes difficult to develop a software incrementally.

2.3.3 Comparison of RAD with Other Models

In this section, we compare the relative advantages and disadvantages of RAD with other life cycle models.

RAD *versus* prototyping model

In the prototyping model, the developed prototype is primarily used by the development team to gain insights into the problem, choose between alternatives, and elicit customer feedback. The code developed during prototype construction is usually thrown away. In contrast, in RAD it is the developed prototype that evolves into the deliverable software.

Though RAD is expected to lead to faster software development compared to the traditional models (such as the prototyping model), the quality and reliability would be inferior.

RAD *versus* iterative waterfall model

In the iterative waterfall model, all the functionalities of a software are developed together. On the other hand, in the RAD model the product functionalities are developed incrementally through heavy code and design reuse. Further, in the RAD model customer feedback is obtained on the developed prototype after each iteration and based on this the prototype is refined. Thus, it becomes easy to accommodate any request for requirements changes. However, the iterative waterfall model does not support any mechanism to accommodate any requirement change requests. The iterative waterfall model does have some important advantages that include the following. Use of the iterative waterfall model leads to production of good quality documentation which can help during software maintenance. Also, the developed software usually has better quality and reliability than that developed using RAD.

RAD *versus* evolutionary model

Incremental development is the hallmark of both evolutionary and RAD models. However, in RAD each increment results in essentially a quick and dirty prototype, whereas in the evolutionary model each increment is systematically developed using the iterative waterfall model. Also in the RAD model, software is developed in much shorter increments compared the evolutionary model. In other words, the incremental functionalities that are developed are of fairly larger granularity in the evolutionary model.

2.4 AGILE DEVELOPMENT MODELS

Over the last two decades or so, projects using iterative waterfall-based life cycle models are becoming increasingly rare due to the rapid shift in the characteristics of the software development projects that are being undertaken. Two changes that are becoming noticeable are rapid shift from development of software products to development of customised software and the increased emphasis and scope for reuse.

As already pointed out, though the iterative waterfall model has been very popular during the 1970s and 1980s, developers face several problems while using it on present-day software projects. The main difficulties include handling change requests from customers during product development, and the unreasonably high cost and time that is incurred while developing customised applications. Capers Jones carried out research involving 800 real-life software development projects, and concluded that on the average 40 per cent of the requirements are arrived well after the development has already begun. In this context, over the last two decades or so, several life cycle models have been proposed to overcome some of the glaring shortcomings of the waterfall-based models that become conspicuous when used in modern software development projects.

In the following, we identify a few reasons why the waterfall-based development was becoming difficult to use in project in recent times:

- In the traditional iterative waterfall-based software development models, the requirements for the system are determined at the start of a development project and are assumed to be fixed from that point on. Later changes to the requirements after the SRS document has been completed are discouraged. If requirement changes become unavoidable later during the development life cycle, then the cost of accommodating these changes becomes prohibitively high. On the other hand, accumulated experience indicates that customers frequently change their requirements during the development period due to a variety of reasons. This is a major source of cost escalation in waterfall-based development projects.
- As pointed out in Chapter 1, over the last two decades or so, customised applications (services) have become common place. The prevalence of customised application development can be gauged from the fact that the sales revenue generated worldwide from services already exceeds that of the software products. Iterative waterfall model is not suitable for development of such software. The main reason for this is that customisation essentially involves reusing most of the parts of an existing application and carrying out only minor modifications to the other parts. For such development projects, the need for appropriate development models was acutely felt, and many researchers started to investigate this issue.
- Waterfall model is called a *heavy weight* model, since there is too much emphasis on producing documentation and usage of tools. This is often a source of inefficiency and causes the project completion time to be much longer in comparison to the customer expectations.
- Waterfall model prescribes almost no customer interactions after the requirements have been specified. In fact, in the waterfall model of software development, customer interactions are largely confined to the project initiation and project completion stages.

The agile software development model was proposed in the mid-1990s to overcome the shortcomings of the waterfall model of development identified above. The agile model could help a project to adapt to change requests quickly.[1] Thus, a major aim of the agile models is to facilitate quick project completion. But, how is agility achieved in these models? Agility is achieved by fitting the process to the project. That is, it gave the required flexibility so that the activities that may not be necessary for a specific project could be easily removed. Also, anything that wastes time and effort is avoided.

Please note that agile model is being used as an umbrella term to refer to a group of development processes. While these processes share certain common characteristics, yet they do have certain subtle differences among themselves. A few popular agile SDLC models are the following:

- Crystal
- Atern (formerly DSDM)
- Feature-driven development
- Scrum
- Extreme programming (XP)
- Lean development
- Unified process

In an agile model, the requirements are decomposed into many small parts that can be incrementally developed. The agile models adopt an incremental and 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 site. No long-term plans are made. The time 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. That is, the delivery date is considered sacrosanct. The development team can, however, decide to reduce the delivered functionality during a time box if necessary.

A central principle of the agile model is the delivery of an increment to the customer after each time box. A few other principles that are central to the agile model are discussed below.

2.4.1 Essential Idea behind Agile Models

For establishing close interactions with the customer during development and to gain a clear understanding of domain-specific issues, each agile project usually includes a customer representative in the team. At the end of each iteration, stakeholders and the customer representative review the progress made and re-evaluate the requirements. The developed increment is installed at the customer site. A distinguishing characteristics of the agile models is frequent delivery of software increments to the customer.

Agile models emphasise use of face-to-face communication in preference over written documents. It is recommended that the development team size be kept small (5-9 people). This would help the team members to meaningfully engage in face-to-face

The agile model emphasises incremental release of working software as the primary measure of progress.

[1] Dictionary meaning of the word agile: To move quickly.

communication and lead to a collaborative work environment. It is implicit that the agile model is suited to the development of small projects. However, can agile development be used for large projects with geographically dispersed team? In this case, the agile model can be used with different teams maintaining as much daily contact as possible through various communication media such as video conferencing, telephone, e-mail, etc.

The following important principles behind the agile model were publicised in the form of a manifesto in 2001:

- Working software over comprehensive documentation.
- Frequent delivery of incremental versions of the software to the customer in intervals of few weeks.
- Requirement change requests from the customer are encouraged and are to be efficiently incorporated.
- Having competent team members and enhancing interactions among them is considered much more important than issues such as usage of sophisticated tools or strict adherence to a documented process. It is advocated that enhanced communication among the development team members can be realised through face-to-face communication rather than through exchange of formal documents.
- Continuous interaction with the customer is considered to be much more important than effective contract negotiation. A customer representative is required to be a part of the development team. This facilitates close, daily co-operation between customers and developers.

Agile development projects usually deploy pair programming.

Several studies indicate that programmers working in pairs produce compact well-written programs and commit fewer errors as compared to programmers working alone.

In pair programming, two programmers work together at one work station. One types in code while the other reviews the code as it is typed in. The two programmers switch their roles every hour or so.

Advantages and disadvantages of agile methods

The agile methods derive much of their agility by relying on the tacit knowledge of the team members about the development project and informal communications to clarify issues, rather than spending significant amounts of time in preparing formal documents and reviewing them. Though this eliminates some overhead, but lack of adequate documentation may lead to several types of problems, which are as follows:

- Lack of formal documents leaves scope for confusion and important decisions taken during different phases can be misinterpreted at later points of time by different team members.
- In the absence of any formal documents, it becomes difficult to get important project decisions such as design decisions to be reviewed by external experts.
- When a project completes and the developers disperse, maintenance can become a problem.

2.4.2 Agile *versus* Other Models

In the following subsections, we compare the characteristics of the agile model with other models of development.

Agile model *versus* iterative waterfall model

The waterfall model is highly structured and makes a project to systematically step through requirements-capture, analysis, specification, design, coding, and testing stages in a planned sequence. Progress is generally measured in terms of the number of completed and reviewed artifacts such as requirements specification document, design documents, test plans, code reviews, etc. In contrast, while using an agile model, progress is measured in terms of the developed and delivered functionalities. In agile model, frequent delivery of working versions of the software is made. However, as regards to similarity it can be said that agile teams use the waterfall model on a small scale, repeating the entire waterfall cycle in every iteration.

If a project that is being developed using waterfall model is cancelled mid-way during development, then there is usually nothing to show from the abandoned project beyond several documents. With agile model however, even if a project is cancelled midway, it still leaves the customer with some worthwhile code, that might possibly have already been put into live operation.

Agile *versus* exploratory programming

Though a few similarities do exist between the agile and exploratory program development styles, there are vast differences between the two as well. Agile development model's frequent re-evaluation of plans, emphasis on face-to-face communication, and relatively sparse use of documentation are similar to that of the exploratory style. Agile teams, however, do follow defined and disciplined processes and carry out systematic requirements capture and rigorous designs, as compared to the chaotic coding that takes place in exploratory programming.

Agile model *versus* RAD model

The important differences between the agile and the RAD models are the following:

- Agile model does not recommend developing prototypes, but emphasises systematic development of each incremental feature. In contrast, the central theme of RAD is to design quick-and-dirty prototypes, which are then refined into production quality code.
- Agile projects logically break down the solution into features that are incrementally developed and delivered. The RAD approach does not recommend this. Instead, developers using the RAD model focus on developing all the features of an application by first doing it badly and then successively improving the code over time.
- Agile teams only demonstrate completed work to the customer. In contrast, RAD teams demonstrate to customers screen mock ups, and prototypes, that usually make several simplifications such as table look-ups rather than performing actual computations.

2.4.3 Extreme Programming Model

Extreme programming (XP) is an important process model under the agile umbrella and was proposed by Kent Beck in 1999. The name of this model reflects the fact that it

recommends taking the *best practices* that have worked well in the past in projects to extreme levels. This model is based on a rather simple philosophy: "If something is known to be beneficial, why not put it to constant use?" Based on this principle, it puts forward several key practices that need to be practiced to the extreme. Please note that most of the key practices that it emphasises were already recognised as good practices for quite some time.

Good practices that need to be carried on to the extreme

In the following subsections, we mention some of the good practices that have been recognized in the extreme programming model and the ways that have been suggested to maximize their use.

Code review: Code review is good since it helps to detect and correct problems most efficiently. XP suggests *pair programming* as the way to achieve continuous review. In pair programming, coding is carried out by pairs of programmers. In each pair, the two programmers take turn in writing code. While one writes code, the other reviews the code being written.

Testing: Testing code helps to remove bugs and improves its reliability. XP suggests *Test-Driven Development* (TDD) to continually write and execute test cases. In the TDD approach, test cases are written for a feature to be implemented, even before any code for it is written. That is, before starting to write code to implement a feature, test cases are written based on user stories. Writing of a piece of code is considered to be complete only after it passes these tests.

Incremental development: Incremental development is good, since it helps to get customer feedback, and extent of features delivered is a reliable indicator of progress. It suggests that the team should come up with new increments every few days.

Simplicity: Simplicity leads to good quality code. It also makes it easier to test and debug the code. Therefore, one should try to create the simplest code and make the basic functionality being written to work. For creating the simplest code, one should ignore the aspects such as efficiency, reliability, maintainability, etc. Once the simplest code works, other desirable aspects can be introduced through refactoring.

Design: Since having a good quality design is important to develop a good quality solution, every team member should perform some design on a daily basis. This can be achieved through *refactoring*, which involves improving a piece of working code for enhanced efficiency and maintainability.

Integration testing: Integration testing is important since it helps to identify the bugs at the interfaces of different functionalities. To this end, extreme programming suggests that the developers should achieve continuous integration. They should perform system builds as well as integration testing several times a day.

Basic idea of extreme programming model

XP is based on frequent releases (called *iterations*), during which the developers implement "user stories". User stories are similar to use cases, but are more informal and are simpler. A user story is the conversational description by the user about a feature of the required system. For example, a set of user stories about a library software can be:

- A library member can issue a book.
- A library member can query about the availability of a book.
- A library member should be able to return a borrowed book.

A user story is a simplistic statement of a customer about a functionality he needs. It does not mention finer details such as the different scenarios that can occur, the precondition to be satisfied before the feature can be invoked, etc.

On the basis of user stories, the project team proposes "metaphors"—a common vision of how the system would work. This essentially is the architectural design (also known as *high-level design*). The development team may decide to construct a *spike* for some feature. A *spike*, is a very simple program that is constructed to explore the suitability of a solution being proposed. A spike can be considered to be similar to a prototype.

XP prescribes several basic activities to be part of the software development process. We discuss these activities in the following subsections:

Coding: XP argues that code is the crucial part of any system development process, since without code it is not possible to have a working system. Therefore, utmost care and attention need to be placed on coding activity. However, the concept of code as used in XP has a slightly different meaning from what is traditionally understood. For example, coding activity includes drawing diagrams (modelling) that will be transformed to code, scripting a web-based system, and choosing among several alternative solutions.

Testing: XP places high importance on testing and considers it be the primary means for developing a fault-free software.

Listening: The developers need to carefully listen to the customers if they have to develop a good quality software. Programmers may not necessarily have an in-depth knowledge of the specific domain of the system under development. On the other hand, customers usually have this domain knowledge. Therefore, for the programmers to properly understand the required functionalities of the system, they have to listen to the customer.

Designing: Without a proper design, a system implementation becomes too complex and the dependencies within the system become too numerous and it becomes very difficult to comprehend the solution. This makes any maintenance activity prohibitively expensive. A good design should result in elimination of complex dependencies within the system. With this intention, effective use of a suitable design technique is emphasised.

Feedback: It espouses the wisdom: "A system staying isolated from the scrutiny of the users is trouble waiting to happen". It recognizes the importance of user feedback in understanding the exact customer requirements. The time that elapses between the development of a version and collection of feedback on it is critical to learning and making changes. It argues that frequent contact with the customer makes the development effective.

XP is in favour of making the solution to a problem as simple as possible. In contrast, the traditional system development methods recommend planning for reusability and future extensibility of code and design at the expense of higher code and design complexity.

Simplicity: A corner-stone of XP is the principle: "build something simple that will work today, rather than trying to build something that would take time and yet may never be used". This in essence means that attention should be

focused on specific features that are immediately needed and making them work, rather than devoting time and energy on speculations about future requirements.

Applicability of extreme programming model

The following are some of the project characteristics that indicate the suitability of a project for development using extreme programming model:

Projects involving new technology or research projects: In this case, the requirements change rapidly and unforeseen technical problems are often needed to be resolved.

Small projects: Extreme programming was proposed in the context of small teams, as face-to-face communication is easier to achieve in this situation.

Project characteristics not suited to development using agile models

The following are some of the project characteristics that indicate unsuitability of agile development model for use in a development project:

- **Stable requirements:** Conventional development models are more suited to use in projects characterised by stable requirements. For such projects, it is known that almost no changes to the gathered requirements will occur. Therefore, process models such as iterative waterfall model that involve making long-term plans during project initiation can meaningfully be used.
- **Mission critical or safety critical systems:** In the development of such systems, the traditional SDLC models are usually preferred to ensure the required reliability.

2.4.4 Scrum

Scrum is one of the agile development models. In the scrum model, the entire project work is divided into small work parts that can incrementally be developed and delivered over time boxes. These time boxes are called *sprints*.

At the end of each sprint, the stakeholders and the team members meet to assess the developed software increment. The stakeholders may suggest any changes and improvements to the developed software that they might feel necessary.

In scrum, the software gets developed over a series sprints. In each sprint, manageable increments to the software (or chunks) are deployed at the client site and the client feedback is obtained after each sprint.

Each sprint is typically 2 to 4 weeks long. During a sprint, an incremental functionality of the software is completed.

Key roles and responsibilities

In the scrum model, the team members assume three basic roles: product owner, scrum master, and team member. The responsibilities associated with these three basic roles are discussed as follows:

- **Product owner:** The product owner represents the customer's perspective and guides the team toward building the right software. In other words, the product owner is responsible for communicating the customer's perspective of the software to the development team. To this end, in every sprint, the product owner in consultation with the team members defines the features of the software to be developed in the next sprint, decides on the release dates, and also may reprioritize the required features that are yet to be developed if necessary.

- **Scrum master:** The scrum master acts as the project manager for the project. The responsibilities of the scrum master include removing any impediments that the project may face, and ensuring that the team is fully productive by fostering close cooperation among the team members. In addition, the scrum master acts as a liaison between the customers, top management, and the team and facilitates the development work. The scrum team is, therefore, shielded by the scrum master from external interferences.
- **Team member:** A scrum team usually consists of cross-functional team members with expertise in areas such as quality assurance, programming, user interface design, and testing. The team is self-organising in the sense that the team members distribute the responsibilities among themselves. This is in contrast to a conventional team where a designated team leader decides who will do what.

Artifacts

Three main artifacts form an important part of the scrum methodology. These are: product backlog, sprint backlog, and sprint burndown chart. We briefly describe the purpose and contents of these three documents in the following subsections.

- **Product backlog:** This document at any time during development contains a prioritized listing of all the features that are yet to be developed. In this document, the features of the software are usually written in the form of user stories. During the course of development, new features may get added to the product backlog and some features may even get deleted. This document is, therefore, a dynamic document that is periodically updated and reprioritised. The product backlog forms the pool of features from which some of the features are chosen for development during the next sprint.
- **Sprint backlog:** Before every sprint, a sprint planning meeting takes place. During a sprint planning meeting, the team identifies one or more features (user stories) from the product backlog and decides on the tasks that would be needed to be undertaken for the development of the identified features and this forms the sprint backlog. In other words, we can say that the sprint backlog lists the tasks that are identified and committed by the team to develop and complete during the ensuing sprint.
- **Sprint burndown chart:** During a sprint, the sprint burndown chart is used as a tool to visualize the progress made and the work remaining to be undertaken on a daily basis. At the end of a team meeting called the daily stand-up meeting, the scrum master determines the work completed in the previous day and the list of work that remains to be completed and represents these in the sprint burndown chart. The sprint burndown chart gives a visual representation of the progress achieved and the work remaining to be undertaken. The horizontal axis of the sprint burndown chart represents the days in the sprint, and the vertical axis shows the amount of work remaining (in hours of effort) at the end of each day. An example sprint burndown chart has been shown in Figure 2.10. As can be seen, the chart visually represents the progress made on a day-to-day basis and the work that is remaining to be completed. This chart, therefore, helps to quickly determine whether the sprint is on schedule and whether all the planned work would finish by the desired date.

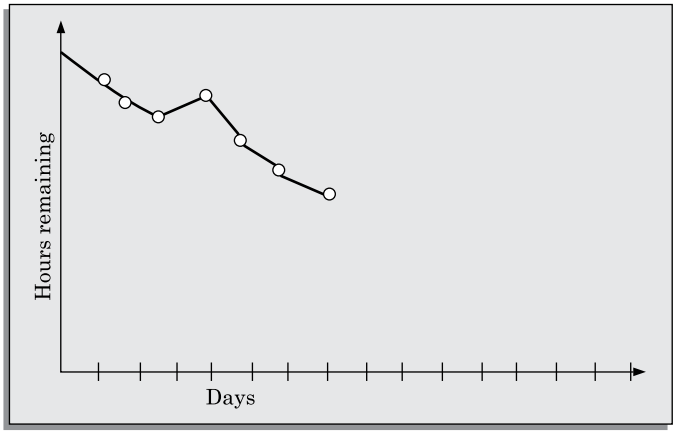


FIGURE 2.10 An example of a sprint burndown chart.

Scrum ceremonies

The term *scrum ceremonies* is used to denote the meetings that are mandatorily held during the duration of a project. The scrum ceremonies include three different types of meetings: sprint planning, daily scrum, and sprint review meeting.

- **Sprint planning:** During the sprint planning meeting, the team members commit to develop and deliver certain features in the ensuing sprint, out of those listed in the product backlog. In this meeting, the product owner works with the team to negotiate which product backlog items the team should work on in the ensuing sprint in order to meet the release goals. It is the responsibility of the scrum master to ensure that the team agrees to realistic goals for a sprint.
- **Daily scrum:** The daily scrum is a short stand-up meeting conducted during every day morning to review the status of the progress achieved so far and the major issues being faced on a day-to-day basis. The daily scrum meeting is not a problem-solving meeting, rather each member updates the teammates about what he/she has achieved in the previous day. Each team member focuses on answering three questions: What did he/she do yesterday? What will he/she do today? What obstacles is he/she facing? The daily scrum meeting helps the scrum master to track the progress made so far and helps to address any problems needing immediate attention. Also the team members get an appraisal of the project status and any specific problems that are being faced by different team members. This meeting is usually short and expected to last only about 15 minutes or 20 minutes. To keep the meeting short and focused, this meeting is conducted with the team members standing up.
- **Sprint review meeting:** At the end of each sprint, a sprint review is conducted. In this meeting, the team demonstrates the new functionality developed during the sprint that was just completed to the product owner and to the stakeholders. Feedback is collected from the participants of the meeting and these either are taken into account in the next sprint or are added to the product backlog.

2.4.5 Lean Software Development

The origin of the Lean software development process can be traced to the Lean process that was developed in a car manufacturing industry (Toyota production system). It was soon found to be an effective product development process that holds significant benefits. It soon became widely accepted and found use in many other types of product manufacturing industries. The central theme of Lean is to achieve overall process efficiency through elimination of various things that cause waste of work and introduce delays. Kanban methodology was developed as a part of the Lean approach to visualize the work flow. Using Kanban, it becomes easy to identify and eliminate potential delays and bottlenecks. Due to its obvious advantages, the Kanban methodology has now become widely accepted and has emerged as a popular methodology on its own standing and is being used in conjunction with other agile methodologies such as XP.

Lean methodology in software development

About a decade back, Lean methodology was adopted for use in the software development projects and has since gained significant popularity. In software development projects, it has been found effective in reducing delays and bottlenecks by minimizing waste. Please note that in the Lean methodology, any work that does not add value to the customer is considered to be waste. A few examples of waste are rework required for defect correction, gold plating and scope creep, delay in staffing, and excessive documentation.

Kanban

Kanban is a Japanese word meaning a sign board. The main objective of Kanban is to provide visibility to the workflow. This in turn is expected to help debottleneck congestions in the process and minimize delays. Though developed in the context of general product manufacturing industries, Kanban has been observed to be highly appropriate in software development projects. A possible reason behind this could be that in manufacturing, the process is visible as the work product progresses along the production line. But, in software development, the visibility of the product, the product line, and the work performed are low. In this context, the Kanban method helps a project team to visualise the workflow, limit work in progress (WIP) at each workflow stage, and measure cycle time.

Kanban board and Kanban card

A principal artifact of the Kanban approach is the Kanban board. A Kanban board is typically a white board on which sticky notes called *Kanban cards* are attached. Kanban cards represent the work items that are waiting for completion at different development stages. A card is moved on the board as the corresponding work is completed at a stage. This provides visualization of the work flow and the work queued up at different stages. The process steps are called *stages* or *work stations*, and are drawn as columns on the Kanban board. On a Kanban board, the work process causing delay is easy to spot, and remedial actions can be initiated. An example Kanban board has been shown in Figure 2.11.

One of the main premise of the Kanban approach is that during development when some work stage becomes overloaded with work, it becomes a bottleneck and introduces delays; thereby slowing down the entire development process. This eventually shows up as schedule slips. Kanban board helps to identify the bottlenecks in a development process

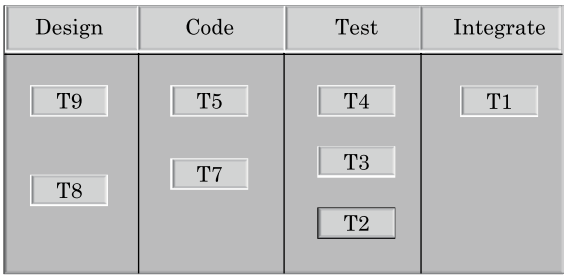


FIGURE 2.11 An example Kanban board.

early, so that corrective actions can be taken in time. In the Kanban board, the different stages of development such as coding, code review, and testing are represented by different columns in the board. The work itself is broken down into small work items and the work item name is written on a card. The card is stuck to the appropriate column of the Kanban board to represent the stage at which the work item is waiting for completion. As the work item progresses through different stages, the card is moved accordingly on the Kanban board. In Kanban, it is a requirement that the number of work items (represented as cards) that can be in progress at any point of time is limited. We can, therefore, say that Kanban enforces work in progress limits (WIP). The general idea behind it is to reduce the workload at a processing station, so that the developers can concentrate only on a few tasks and complete those before taking up anything new. The goal of limiting WIP is to reduce stress and to raise productivity as well as product quality.

Visibility provided by Kanban board

The Kanban board provides visibility to the software process. It shows work assigned to each developer at any time and the work flow. It, therefore, becomes possible to define and communicate the priorities and the bottlenecks get highlighted. The average time it takes to complete a work item (sometimes called the *cycle time*) is tracked and optimized so that the process becomes efficient and predictable. The Kanban system, therefore, helps to limit the amount of work in process so that the work flowing through the system matches its full capacity.

2.5 SPIRAL MODEL

This model gets its name from the appearance of its diagrammatic representation that looks like a spiral with many loops (see Figure 2.12). The exact number of loops of the spiral is not fixed and can vary from project to project. The number of loops shown in Figure 2.12 is just an example. Each loop of the spiral is called a *phase* of the software process. The exact number of phases through which the product is developed can be varied by the project

While the prototyping model does provide explicit support for risk handling, the risks are assumed to have been identified completely before the project start. This is required since the prototype is constructed only at the start of the project. In contrast, in the spiral model prototypes are built at the start of every phase. Each phase of the model is represented as a loop in its diagrammatic representation. Over each loop, one or more features of the product are elaborated and analysed and the risks at that point of time are identified and are resolved through prototyping. Based on this, the identified features are implemented.

manager depending upon the project risks. A prominent feature of the spiral model is handling unforeseen risks that can show up much after the project has started. In this context, please recollect that the prototyping model can be used effectively only when the risks in a project can be identified upfront before the development work starts. As we shall discuss, this model achieves this by incorporating much more flexibility compared to SDLC of other models.

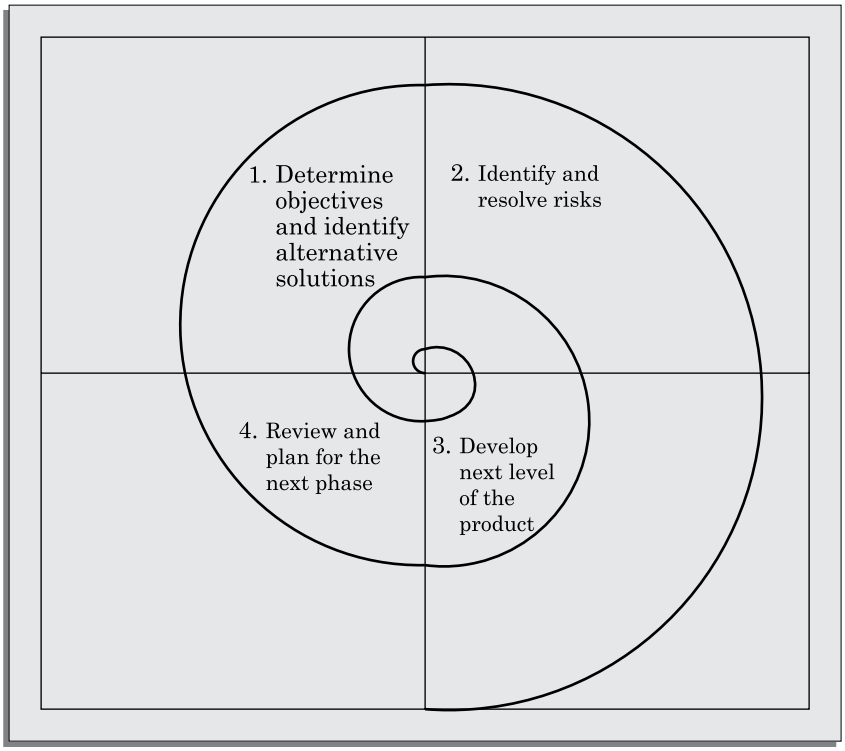


FIGURE 2.12 Spiral model of software development.

Risk handling in spiral model

A risk is essentially any adverse circumstance that might hamper the successful completion of a software project. As an example, consider a project for which a risk can be that data access from a remote database might be too slow to be acceptable by the customer. This risk can be resolved by building a prototype of the data access subsystem and experimenting with the exact access rate. If the data access rate is too slow, possibly a caching scheme can be implemented or a faster communication scheme can be deployed to overcome the slow data access rate. Such risk resolutions are easier done by using a prototype as the pros and cons of an alternate solution scheme can evaluated faster and inexpensively, as compared to experimenting using the actual software application being developed. The spiral model supports coping up with risks by providing the scope to build a prototype at every phase of software development.

2.5.1 Phases of the Spiral Model

Each phase in this model is split into four sectors (or quadrants) as shown in Figure 2.12. In the first quadrant, a few features of the software are identified to be taken up for immediate development based on how crucial it is to the overall software development. With each iteration around the spiral (beginning at the center and moving outwards), progressively more complete versions of the software get built. In other words, implementation of the identified features forms a phase.

Quadrant 1: The objectives are investigated, elaborated, and analysed. Based on this, the risks involved in meeting the phase objectives are identified. In this quadrant, alternative solutions possible for the phase under consideration are proposed.

Quadrant 2: During the second quadrant, the alternative solutions are evaluated to select the best possible solution. To be able to do this, the solutions are evaluated by developing an appropriate prototype.

Quadrant 3: Activities during the third quadrant consist of developing and verifying the next level of the software. At the end of the third quadrant, the identified features have been implemented and the next version of the software is available.

Quadrant 4: Activities during the fourth quadrant concern reviewing the results of the stages traversed so far (i.e. the developed version of the software) with the customer and planning the next iteration of the spiral.

In the spiral model of development, the project manager dynamically determines the number of phases as the project progresses. Therefore, in this model, the project manager plays the crucial role of tuning the model to specific projects.

The radius of the spiral at any point represents the cost incurred in the project so far, and the angular dimension represents the progress made so far in the current phase.

To make the model more efficient, the different features of the software that can be developed simultaneously through parallel cycles are identified. To keep our discussion simple, we shall not delve into parallel cycles in the spiral model.

Advantages/pros and disadvantages/cons of the spiral model

There are a few disadvantages of the spiral model that restrict its use to only a few types of projects. To the developers of a project, the spiral model usually appears as a complex model to follow, since it is risk-driven and is more complicated phase structure than the other models we discussed. It would therefore be counterproductive to use, unless there are knowledgeable and experienced staff in the project. Also, it is not very suitable for use in the development of outsourced projects, since the software risks need to be continually assessed as it is developed.

In spite of the disadvantages of the spiral model that we pointed out, for certain categories of projects, the advantages of the spiral model can outweigh its disadvantages.

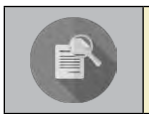
In this regard, it is much more powerful than the prototyping model. Prototyping model can meaningfully

For projects having many unknown risks that might show up as the development proceeds, the spiral model would be the most appropriate development model to follow.

be used when all the risks associated with a project are known beforehand. All these risks are resolved by building a prototype before the actual software development starts.

Spiral model as a meta model

As compared to the previously discussed models, the spiral model can be viewed as a *meta model*, since it subsumes all the discussed models. For example, a single loop spiral actually represents the waterfall model. The spiral model uses the approach of the prototyping model by first building a prototype in each phase before the actual development starts. This prototypes are used as a risk reduction mechanism. The spiral model incorporates the systematic step-wise approach of the waterfall model. Also, the spiral model can be considered as supporting the evolutionary model—the iterations along the spiral can be considered as evolutionary levels through which the complete system is built. This enables the developer to understand and resolve the risks at each evolutionary level (i.e. iteration along the spiral).



CASE STUDY 2.2

Galaxy Inc. undertook the development of a satellite-based communication between mobile handsets that can be anywhere on the earth. In contrast to the traditional cell phones, by using a satellite-based mobile phone a call can be established as long as both the source and destination phones are in the coverage areas of some base stations. The system would function through about six dozen satellites orbiting the earth. The satellites would directly pick up the signals from a handset and beam signal to the destination handset. Since the foot prints of the revolving satellites would cover the entire earth, communication between any two points on the earth, even between remote places such as those in the Arctic Ocean and Antarctica, would also be possible. However, the risks in the project are many, including determining how the calls among the satellites can be handed-off when they are themselves revolving at a very high speed. In the absence of any published material and availability of staff with experience in development of similar products, many of the risks cannot be identified at the start of the project and are likely to crop up as the project progresses. The software would require several million lines of code to be written. Galaxy Inc. decided to deploy the spiral model for software development after hiring highly qualified staff. To speed up the software development, independent parts of the software were developed through parallel cycles on the spiral. The cost and delivery schedule were refined many times, as the project progressed. The project was successfully completed after five years from start date.

2.6 A COMPARISON OF DIFFERENT LIFE CYCLE MODELS

The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model cannot be used in practical development projects, since this model supports no mechanism to correct the errors that are committed during any of the phases but detected at a later phase. This problem is overcome by the iterative waterfall model through the provision of feedback paths.

The iterative waterfall model is probably the most widely used software development model so far. This model is simple to understand and use. However, this model is suitable

only for well-understood problems, and is not suitable for development of very large projects and projects that suffer from large number of risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood, however all the risks can be identified before the project starts. This model is especially popular for development of the user interface part of projects.

The evolutionary approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also used widely for object-oriented development projects. Of course, this model can only be used if incremental delivery of the system is acceptable to the customer.

The spiral model is considered a *meta model* and encompasses all other life cycle models. Flexibility and risk handling are inherently built into this model. The spiral model is suitable for development of technically challenging and large software that are prone to several kinds of risks that are difficult to anticipate at the start of the project. However, this model is much more complex than the other models—this is probably a factor deterring its use in ordinary projects.

Let us now compare the prototyping model with the spiral model. The prototyping model can be used if the risks are few and can be determined at the start of the project. The spiral model, on the other hand, is useful when the risks are difficult to anticipate at the beginning of the project, but are likely to crop up as the development proceeds.

Let us compare the different life cycle models from the viewpoint of the customer. Initially, customer confidence is usually high on the development team irrespective of the development model followed. During the lengthy development process, customer confidence normally drops off, as no working software is yet visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment. On the other hand, an evolutionary approach lets the customer experiment with a working software much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer's trauma of getting used to an entirely new system. The gradual introduction of the software via incremental phases provides time to the customer to adjust to the new software. Also, from the customer's financial viewpoint, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

2.6.1 Selecting an Appropriate Life Cycle Model for a Project

We have discussed the advantages and disadvantages of the various life cycle models. However, how to select a suitable life cycle model for a specific project? The answer to this question would depend on several factors. A suitable life cycle model can possibly be selected based on an analysis of issues such as the following:

Characteristics of the software to be developed: The choice of the life cycle model to a large extent depends on the nature of the software that is being developed. For small services projects, the agile model is favoured. On the other hand, for product and embedded software development, the iterative waterfall model can be preferred. An evolutionary model is a suitable model for object-oriented development projects.

Characteristics of the development team: The skill-level of the team members is a significant factor in deciding about the life cycle model to use. If the development team is experienced in developing similar software, then even an embedded software can be developed using an iterative waterfall model. If the development team is entirely novice, then even a simple data processing application may require a prototyping model to be adopted.

Characteristics of the customer: If the customer is not quite familiar with computers, then the requirements are likely to change frequently as it would be difficult to form complete, consistent, and unambiguous requirements. Thus, a prototyping model may be necessary to reduce later change requests from the customers.

SUMMARY

- During the development of any type of software, adherence to a suitable process model has become universally accepted by software development organisations. Adoption of a suitable life cycle model is now accepted as a primary necessity for successful completion of projects.
- We discussed only the central ideas behind some important process models. Good software development organisations carefully and elaborately document the precise process model they follow and typically include the following in the document:
  - Identification of the different phases.
  - Identification of the different activities in each phase and the order in which they are carried out.
  - The phase entry and exit criteria for different phases.
  - The methodology followed to carry out the different activities.
- Adherence to a software life cycle model encourages the team members to perform various development activities in a systematic and disciplined manner. It also makes management of software development projects easier.
- The principle of detecting errors as close to their point of introduction as possible is known as phase containment of errors. Phase containment minimises the cost to fix errors.
- The classical waterfall model can be considered as the basic model and all other life cycle models are embellishments of this model. Iterative waterfall model has been the most widely used life cycle model so far, though agile models have gained prominence lately.
- Different life cycle models have their own advantages and disadvantages. Therefore, an appropriate life cycle model should be chosen for the problem at hand. After choosing a basic life cycle model, software development organisations usually tailor the standard life cycle models according to their needs.
- Even though an organisation may follow whichever life cycle model that may appear appropriate to a project, the final document should reflect as if the software was developed using the classical waterfall model. This makes it easier for the maintainers to understand the software documents.

EXERCISES

MULTIPLE CHOICE QUESTIONS

Choose the correct option for each of the following questions:

1. Which one of the following may be experienced by a software development team when it adopts a systematic development process model in preference to a build-and-fix style of development?
(a) Increased documentation overhead
(b) Increased development cost
(c) Decreased maintainability
(d) Increased development time

2. A software process model represents which one of the following:
(a) The way in which software is developed
(b) The way in which software processes data
(c) The way in which software is used
(d) The way in which software may fail

3. Prototyping life cycle model is appropriate when a project suffers from which one of the following risks:
(a) Schedule slippage
(b) Manpower turnover
(c) Incomplete and uncertain requirements
(d) Poor quality of outsourced work

4. Which one of the following activity spans all stages of a software development life cycle (SDLC)?
(a) Coding
(b) Testing
(c) Project management
(d) Design

5. The operation phase of the waterfall model is a synonym for which one of the following phases:
(a) Coding and unit testing phase
(b) Integration and system testing phase
(c) Maintenance phase
(d) Design phase

6. The implementation phase of the waterfall model is a synonym for which one of the following phases:
(a) Coding and unit testing phase
(b) Integration and system testing phase
(c) Maintenance phase
(d) Design phase

7. Unit testing is carried out during which phase of the waterfall model:
(a) Implementation phase

(b) Testing phase
(c) Maintenance phase
(d) Design phase

8. Which one of the following phases accounts for the maximum effort during development of a typical software?
(a) Coding
(b) Testing
(c) Designing
(d) Specification

9. Which one of the following is not a standard software development process model?
(a) Waterfall Model
(b) Recursive Model
(c) RAD Model
(d) V-Model

10. Which one of the following feedback paths is not present in an iterative waterfall model?
(a) Design phase to feasibility study phase
(b) Implementation phase to design phase
(c) Implementation phase to requirements specification phase
(d) Design phase to requirements specification phase

11. Which one of the following can be considered to be a suitable SDLC model for developing a moderate- sized software for which the customer is not clear about his exact requirements?
(a) RAD model
(b) V-model
(c) Iterative waterfall model
(d) Classical waterfall model

12. Which one of the following SDLC models would be suitable for use in a project involving customisation of a computer communication package? Assume that the project would be manned by experienced personnel and that the schedule for the project has been very aggressively set?
(a) Spiral model
(b) Iterative waterfall model
(c) RAD model
(d) Agile model

13. Which one of the following life cycle models lacks the characteristics of iterative software development?
(a) Spiral model
(b) Prototyping model
(c) Classical waterfall model
(d) Evolutionary model

14. Which one of the following life cycle models does not involve constructing a prototype any time during software development?

(a) Spiral model
(b) Prototyping model
(c) RAD model
(d) Evolutionary model

15. The prototyping model may not be appropriate for a project for developing a gaming software when it has which one of the following characteristics?
(a) Unresolved technical issues
(b) Unclear user requirements
(c) Significant GUI components
(d) Significant business risk arising from the fact that several competitors may in the near future come out with similar gaming software

16. Which one of the following is not a characteristic of the agile model of software development?
(a) Completion of requirements specification before design phase
(b) Evolutionary development
(c) Iterative development
(d) Periodic delivery of working software

17. Which one of the following SDLC models can be considered to be most effective for determination of the exact customer requirements when the customer is not clear about the requirements?
(a) Iterative waterfall model
(b) V-model
(c) Prototyping model
(d) Classical waterfall model

18. Change requests from customers later in the development cycle are easiest to handle in which one of the following life cycle models:
(a) Iterative waterfall model
(b) Prototyping model
(c) V-model
(d) Evolutionary model

19. Assume that you are the project manager of a project for developing a data processing application in which the user requirements for the GUI part are not very clear. Which one of the following life cycle models would you use to develop the GUI part?
(a) Classical waterfall model
(b) Iterative waterfall model
(c) Prototyping model
(d) Spiral model

20. The angular dimension of the spiral model does not represent which one of the following?
(a) Cost incurred so far
(b) Number of features implemented so far
(c) Progress in the implementation of the current feature
(d) Number of risks that have been resolved so far

21. The radial dimension of the spiral model represents which one of the following:
(a) Cost incurred so far
(b) Number of features implemented so far
(c) Progress in the implementation of the current feature
(d) Number of risks that have been resolved so far

22. Which one of the following activities is not carried out during the testing phase of the waterfall life cycle model?
(a) Unit testing
(b) Integration testing
(c) System testing
(d) Debugging

23. Which one of the following is not a phase of the waterfall life cycle model?
(a) Feasibility study
(b) Project management
(c) Maintenance
(d) Requirements specification

24. Which one of the following life cycle models has the least similarity with the waterfall model?
(a) Prototyping
(b) V-model
(c) Spiral
(d) Scrum

25. Presence of which one of the following characteristics in a project would indicate the suitability of the extreme programming (XP) model:
(a) Stable and well understood requirements
(b) Mission critical software
(c) Service-oriented project
(d) Extremely large project

26. Which one of the following life cycle models does not involve constructing a prototype any time during software development?
(a) Spiral model
(b) Prototyping model
(c) RAD
(d) Evolutionary model

27. Which one of the following is the most appropriate reason behind scrum teams to be called self-organizing?
(a) The team members independently decide when to join and leave a project
(b) The team members decide among themselves their respective roles and responsibilities
(c) The team members decide among themselves their respective roles in the project, after which the scrum master assigns them specific responsibilities.
(d) The team members decide who would report to whom among themselves.

28. Which one of the following is not a sprint ceremony in scrum?
(a) Sprint planning meeting
(b) Daily stand-up meeting
(c) Sprint retrospective meeting
(d) Project start up meeting

29. The prototyping model essentially involves incorporating the construction of a prototype in the iterative waterfall process model. The prototype is constructed in which one of the following phases of the waterfall model of software development:
(a) Feasibility study
(b) Requirements analysis and specification
(c) Design
(d) Coding and unit testing

30. Which one of the following is true for prototypes constructed in the prototyping model of software development?
(a) Construction of prototypes incurs significant cost and should be avoided as far as possible.
(b) A prototype is useful for eliciting early feedback on requirements for the stakeholders.
(c) Prototypes are usually constructed toward the end of the design phase.
(d) Prototypes increase the risk of misunderstandings between developers and users.

31. In the prototyping model of software development, which one of the following is the principal reason for developing a prototype?
(a) A prototype is iteratively enhanced into a working software and is finally deployed at the client site
(b) A prototype is deployed at the client site as an operational software till the full version is developed
(c) A working prototype helps the customer to provide meaningful feedback about requirements
(d) A prototype helps in development of accurate project schedule

32. Which one of the following is not an expected outcome of using Kanban method?
(a) Reduced delay
(b) Reduced fatigue of team members
(c) Reduced manpower
(d) More balanced work load

33. In which one of the following SDLC models, testing activities are spread over the entire development life cycle?
(a) Iterative waterfall model
(b) V-model
(c) Prototyping model
(d) Classical waterfall model

34. Which of the following is the principal reason for developing a prototype?
(a) It can be used as an early production tool
(b) It may solve a problem that is not included in the requirements

(c) It allows the customer to provide feedback about requirements
(d) It reduces the schedule for development through alpha testing

35. Which one of the following is not a characteristic of the agile model of software development?
(a) Well demarcated phases
(b) Evolutionary development
(c) Iterative development
(d) Periodic delivery of working software

36. Which one of the following is expected to be achieved by pair-wise programming and is not achieved by traditional programming?
(a) Incremental development
(b) Test-driven development
(c) Continuous review
(d) Iterative development

37. Defects get introduced into a work product due to mistakes committed by the members of the development team. A mistake committed during which one of the following phases and detected after product release is usually the most expensive to correct:
(a) System testing
(b) Design
(c) Coding and unit testing
(d) Requirements analysis and specification

38. Which one of the following is not a characteristic of the agile software development model?
(a) Long-term planning
(b) Evolutionary development
(c) Iterative development
(d) Periodic delivery of working software

39. Pair programming intends to take which one of the following best practices to the extreme:
(a) Design
(b) Incremental development
(c) Code review
(d) Simplicity

40. Which one of the following phrases would most appropriately to complete the sentence? "The later a defect is found, _____ ."
(a) the easier it is to find
(b) the more expensive it is to repair
(c) the less important it is to the product
(d) the faster it is to fix

41. For a typical software product, testing should accounts for what percentage of the total software development costs:
(a) 10–20
(b) 40–50
(c) 70–80
(d) 5–10

42. Which one of the following is not an agile software development methodology?
(a) Extreme programming (XP)
(b) Scrum
(c) Lean software development
(d) Rapid application development (RAD)

43. In the spiral model, when is risk analysis performed?
(a) In the first loop
(b) In the last loop
(c) In every loop
(d) Before the first loop

TRUE OR FALSE

State whether the following statements are **TRUE** or **FALSE**. Give reasons behind your answers.

1. If the phase containment of errors principle is not followed during software development, then development cost would increase.

2. Evolutionary life cycle model would be appropriate to develop a software that appears to be beset with a large number of risks.

3. The number of phases in the spiral life cycle model is not fixed and is normally determined by the project managers as the project progresses.

4. The primary purpose of phase containment of errors is to develop an error-free software.

5. Development of a software using the prototyping life cycle model is always more expensive than development of the same software using the iterative waterfall model due to the additional cost incurred to construct a throw-away prototype.

6. When a large software is developed by a commercial software development house using the iterative waterfall model, there do not exist precise points of time at which transitions from one phase to another take place.

7. Among all phases of software development, an undetected error from the design phase that ultimately gets detected during the system acceptance test costs the maximum.

8. If a team developing a moderate sized software product does not care about phase containment of errors, it can still produce a reliable software, *al beit* at a higher cost compared to the case where it attempts phase containment of errors.

9. The angular dimension in a spiral model of software development indicates the total cost incurred in the project till that time.

10. When the spiral model is used in a software development project, the number of loops in the spiral is fixed by the project manager during the project planning stage.

11. RAD would be a suitable life cycle model for developing a commercial operating system.

12. RAD is a suitable process model to use for developing a safety-critical application such as a controller for a nuclear reactor.

REVIEW QUESTIONS

1. What do you understand by the term *software life cycle*? Why is it necessary to model software life cycle and to document it?
2. What do you understand by the term *software development life cycle model (SDLC)*? What problems might a software development organization face, if it does not follow any SDLC model for development of a large-sized software?
3. What problems would a software development team face if it does not have a documented process model, and therefore, the project teams follow only informal ones?
4. Are the terms SDLC and software development process synonymous? Explain your answer.
5. Why is it important for an organization to properly document its development process?
6. Consider software development by a project team:
   (a) Mention the major activities that are undertaken during the development of a software.
   (b) Name an activity that spans all the development phases.
7. What do you mean by a software development process? What is the difference between a methodology and a process? Explain your answer using a suitable example.
8. Which are the major phases of the waterfall model of software development? Which phase consumes the maximum effort for developing a typical software?
9. Why is the classical waterfall model called an idealistic development model? Does this model of development has any practical use at all? Explain your answer.
10. Consider the following assertion: "The classical waterfall model is an idealistic model." Based on this assertion, answer the following:
    (a) Justify why the above assertion is true.
    (b) Even if the classical waterfall model is an idealistic model, is there any practical use of this model at all? Explain your answer.
11. What is the difference between programming-in-the-small and programming-in-the-large? Is using waterfall SDLC model a good idea for a programming-in-the-small project? Explain your answer.
12. Draw a schematic diagram to represent the iterative waterfall model of software development. On your diagram represent the following:
    (a) The phase entry and exit criteria for each phase.
    (b) The deliverables that need to be produced at the end of each phase.
13. What are the objectives of the feasibility study phase of software development? Explain the important activities that are carried out during the feasibility study phase of a software development project. Who carries out these activities? Mention suitable phase entry and exit criteria for this phase.
14. Give an example of a software development project for which the iterative waterfall model is not suitable. Briefly justify your answer.

15. In a real-life software development project using iterative waterfall SDLC, is it a practical necessity that the different phases overlap? Explain your answer and the effort distribution over different phases.

16. Identify five reasons as to why the customer requirements may change after the requirements phase is complete and the SRS document has been signed off.

17. Identify the criteria based on which a suitable life cycle model can be chosen for a given software development project. Illustrate your answer using suitable examples.

18. Briefly explain the important differences and similarities between the incremental and evolutionary models of SDLCs.

19. What do you understand by "build-and-fix" style of software development? Diagrammatically depict the typical activities in this style of development and their ordering. Identify at least four major problems that would arise, if a large professional software development project is undertaken by a project team using a "build-and-fix" style of software development.

20. What do you understand by the "99 per cent complete" syndrome that software project managers sometimes face? What are its underlying causes? What problems does it create for project management? What are its remedies?

21. While using the iterative waterfall model to develop a commercial software for a business application, discuss how the effort spent on the different phases is spread over time.

22. Which life cycle model would you follow for developing software for each of the following applications? Mention the reasons behind your choice of a particular life cycle model.

(a) A well-understood data processing application.
(b) A new software that would connect computers through satellite communication. Assume that your team has no previous experience in developing satellite communication software.
(c) A software that would function as the controller of a telephone switching system.
(d) A new library automation software that would link various libraries in the city.
(e) An extremely large software that would provide, monitor, and control cellular communication among its subscribers using a set of revolving satellites.
(f) A new text editor.
(g) A compiler for a new language.
(h) An object-oriented software development effort.
(i) The graphical user interface part of a large software.

23. Briefly explain the V SDLC model and answer the following specific questions pertaining to the V SDLC.

(a) What are the strengths and weaknesses of the V-model?
(b) Outline the similarities and differences of the V-model with the iterative waterfall model.
(c) Give an example of a development project for which V-model can be considered appropriate and also give an example of a project for which it would be clearly inappropriate.

24. Briefly explain the V SDLC model. Identify why for developing safety-critical software, the V SDLC model is usually considered suitable.

25. With respect to the prototyping model for software development, answer the following:
(a) What is a prototype?
(b) Is it necessary to develop a prototype for all types of projects?
(c) If you answer to part (b) of the question is no, then mention under what circumstances is it beneficial to construct a prototype.
(d) If your answer to part (b) of the question is yes, then explain does construction of a prototype always increase the overall cost of software development.

26. If the prototyping model is being used by a team developing a moderate sized software, is it necessary to develop an SRS document? Justify your answer.

27. Consider that a software development project that is beset with many risks. But, assume that it is not possible to anticipate all the risks in the project at the start of the project and some of the risks can only be identified much after the development is underway. As the project manager of this project, would you recommend the use of the prototyping or the spiral model? Explain your answer.

28. What are the major advantages of first constructing a working prototype before starting to develop the actual software? What are the disadvantages of this approach?

29. Explain how a software development effort is initiated and finally terminated in the spiral model.

30. Suppose a travel agency needs a 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?

31. Explain why the spiral life cycle model is considered to be a meta model.

32. Both the prototyping model and the spiral model have been designed to handle risks. Identify how exactly risk is handled in each. How do these two models compare with each other with respect to their risk handling capabilities?

33. Explain using a suitable example, the types of software development for which the spiral model is suitable. Is the number of loops of the spiral fixed for different development projects? If not, explain how the number of loops in the spiral is determined.

34. Discuss the relative merits of developing a throw-away prototype to resolve risks versus refining a developed prototype into the final software.

35. Answer the following questions, using one sentence for each:
(a) How are the risks associated with a project handled in the spiral model of software development?
(b) Which types of risks are be better handled using the spiral model compared to the prototyping model?
(c) Give an example of a project where the spiral model can be meaningfully be deployed.

36. Compare the relative advantages of using the iterative waterfall model and the spiral model of software development for developing an MIS application. Explain with the help of one suitable example each, the types of projects for which you would use the waterfall model of software development, and the types of projects for which you would use the spiral model.

37. Briefly discuss the evolutionary process model. Explain using a suitable example, the software development project for which the evolutionary life cycle model is suitable. Compare the advantages and disadvantages of this model with the iterative waterfall model.

38. Assume that a software development company is already experienced in developing payroll software and has developed similar software for several customers (organisations). Assume that the software development company has received a request from a certain customer (organisation), which was still using manually processing of its payrolls. For developing a payroll software for this organisation, which life cycle model should be used? Justify your answer.

39. Explain why it may not be prudent to use the spiral model in the development of any large software.

40. Instead of having a one time testing of a software at the end of its development, why are three different levels of testing—unit testing, integration testing, and system testing—are necessary? What is the main purpose of each of these different levels of testing?

41. What do you understand by the term phase containment of errors? Why phase containment of errors is considered to be important? How can phase containment of errors be achieved in a software development project?

42. Irrespective of whichever life cycle model is followed for developing a software, why is it necessary for the final documents to describe the software as if it were developed using the classical waterfall model?

43. What are the major shortcomings of the iterative waterfall model? Name the life cycle models that overcome any of your identified shortcomings. How are the shortcomings overcome in those models?

44. For which types of development projects is the V-model appropriate? Briefly explain the V-model and point out its strengths and weaknesses.

45. Identify the main motivation and goals behind the development of the RAD model. How does the model help achieve the identified goals?

46. Explain the following aspects of rapid application development (RAD) SDLC model:
(a) What is a time box in a RAD model?
(b) How does RAD facilitate faster development?
(c) Identify the key differences between the RAD model and the prototyping model.
(d) Identify the types of projects for which RAD model is suitable and the types for which it is not suitable.

47. Suggest a suitable life-cycle model for a software project which your organisation has undertaken on behalf of certain customer who is unsure of his requirements and is likely to change his requirements frequently, since the business process of the

customer (organisation) is of late changing rapidly. Give the reasons behind your answer.

48. Draw a labelled schematic diagram to represent the spiral model of software development. Is the number of loops of the spiral fixed? If your answer is affirmative, write down the number of the loops that the spiral has. If your answer is negative, explain how can a project manager determine the number of loops of the spiral.

49. Assume that you are the project manager of a development team that is using the iterative waterfall model for developing a certain software. Would you recommend that the development team should start a development phase only after the previous phase is fully complete? Explain your answer.

50. Identify the major differences between the iterative and evolutionary SDLCs.

51. Explain how the characteristics of the product, the development team, and the customer influence the selection of an appropriate SDLC for a project.

52. With respect to the rapid application development (RAD) model, answer the following:

(a) Explain the different life cycle activities that are carried out in the RAD model.
(b) How does RAD model help accommodate change requests late in the development.
(c) How does RAD help in faster software development.
(d) Give examples of two projects for which RAD would be a suitable model for development.
(e) Point out the advantages and disadvantages of the RAD model as compared to (i) prototyping model and (ii) evolutionary model.
(f) Point a disadvantage of the RAD model as compared to iterative waterfall model.
(g) Identify the characteristics that make a project suited to RAD style of development.
(h) Identify the characteristics that make a project unsuited to RAD style of development.

53. Identify the important factors that influence the choice of a suitable SDLC model for a software development project. Give a few examples to illustrate your answer.

54. Explain the important features of the agile software development model.

(a) Compare the advantages and disadvantages of the agile model with iterative waterfall and the exploratory programming model.
(b) Is the agile life cycle model suitable for development of embedded software? Briefly justify your answer.

55. Briefly explain the agile software development model. Give an example of a project for which the agile model would be suitable and one project for which the agile model would not be appropriate.

56. Explain the similarities in the objectives and practices of the RAD, agile, and extreme programming (XP) models of software development. Also explain the dissimilarities among these three models.

57. Briefly discuss the RAD model. Identify the main advantages of RAD model as compared to the iterative waterfall model. How does RAD model achieve faster development as compared to iterative waterfall model?

58. Compare the relative advantages of RAD, iterative waterfall, and the evolutionary models of software development.

59. Identify and explain the important best practices that have been incorporated in the extreme programming model.

60. Using one or two sentences explain the important shortcomings of the classical waterfall model that RAD, agile, and extreme programming (XP) models of software development address.

61. Briefly explain the extreme programming (XP) SDLC model. Identify the key principles that need to be practiced to the extreme in XP. What is a spike in XP? Why is it required?

62. Identify how the agile SDLCs achieve reductions to the development time and cost. Are there any pitfalls of achieving cost and time reductions this way?

63. Suppose a development project has been undertaken by a company for customising one of its existing software on behalf of a specific customer. Identify two major advantages of using an agile model over the iterative waterfall model.

64. Which life cycle model would you recommend for developing an object-oriented software? Justify your answer.

65. What do you understand by pairwise programming? What are its advantages over traditional programming?

66. Explain RAD life cycle model. Briefly explain how RAD facilitates faster software development and helps to accommodate change requests from the user.

67. Name the agile model that focuses on minimizing wastage in the development process. Briefly explain the artifacts and methodology that it follows to achieve this.

68. Briefly explain the central principles of extreme programming (XP). In the context of XP, explain the following terms: metaphor, spike, and pair programming.

69. In the context of scrum software development model, explain the following terms: sprint, product backlog, sprint backlog, sprint burndown chart.

70. Distinguish between programming-in-the-small and programming-in-the-large. Identify at least one life cycle model suitable for each. Briefly justify your answer.

71. Distinguish between incremental and evolutionary models of development. Agile methods are often referred to as *incremental and evolutionary*. Explain what is meant by this term.

72. What do you understand by heavy weight and light weight software process models? Name one popular model belonging to each category.

73. Mention the development activities that the extreme programming model suggests to take to extreme. Briefly explain the specific ways it suggests to take these activities to extreme during software development.