2.2 CLASSICAL WATERFALL MODEL

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, we shall see that it is not a practical model in the sense that it cannot be used in actual software development projects. That is, we can consider this model to be a *theoretical way of developing software*. Then why

study this model at all? The reason is that all other life cycle models are in some way or other based on the classical waterfall model. We therefore need to first understand the classical waterfall model well, in order to be able to appreciate and develop proper understanding of other life cycle models. Besides, we shall see later in this text that this model though cannot be used for software development; it is the model that is normally adhered to for developing software documentation.

The classical waterfall model divides the life cycle into the phases shown in Figure 2.1. Observe that the diagrammatic representation of this model resembles a cascade of waterfalls. This resemblance possibly justifies the name of this model.

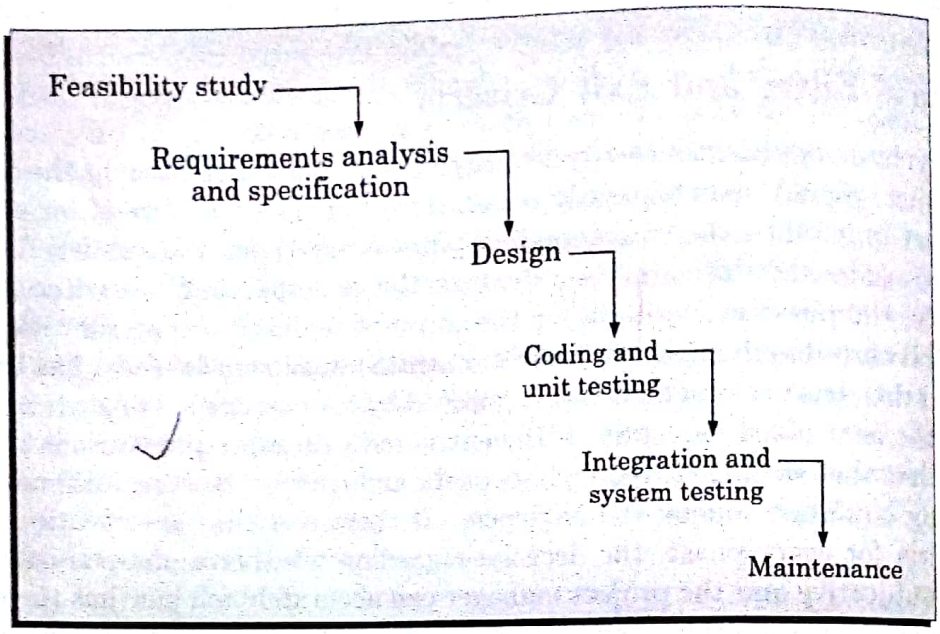

Figure 2.1: Classical waterfall model.

2.2.1 Phases of Classical Waterfall Model

The classical waterfall model breaks down the life cycle into an intuitive set of phases. The different phases of this model 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*. The software is developed during the development phases, and at the end of the development phases of the life cycle, the product becomes ready to be delivered to the customer. The maintenance phase commences after completion of the development phases.

An activity that spans all phases of any software development is project management. Since the project management activity spans the entire project duration, it is not shown separately in Figure 2.1. Even though conveniently omitted in the life cycle diagram, project management nevertheless is an important activity in the life cycle and deals with managing the effort at all stages of product development and maintenance.

The work needed to be carried out during different life cycle phases typically requires relatively different efforts to be put in by the development team. The relative amount of effort necessary for completing the activities of different phases for a typical product are

shown in Figure 2.2. We can 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% of the total life cycle effort 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, we briefly describe the different phases of the classical waterfall model.

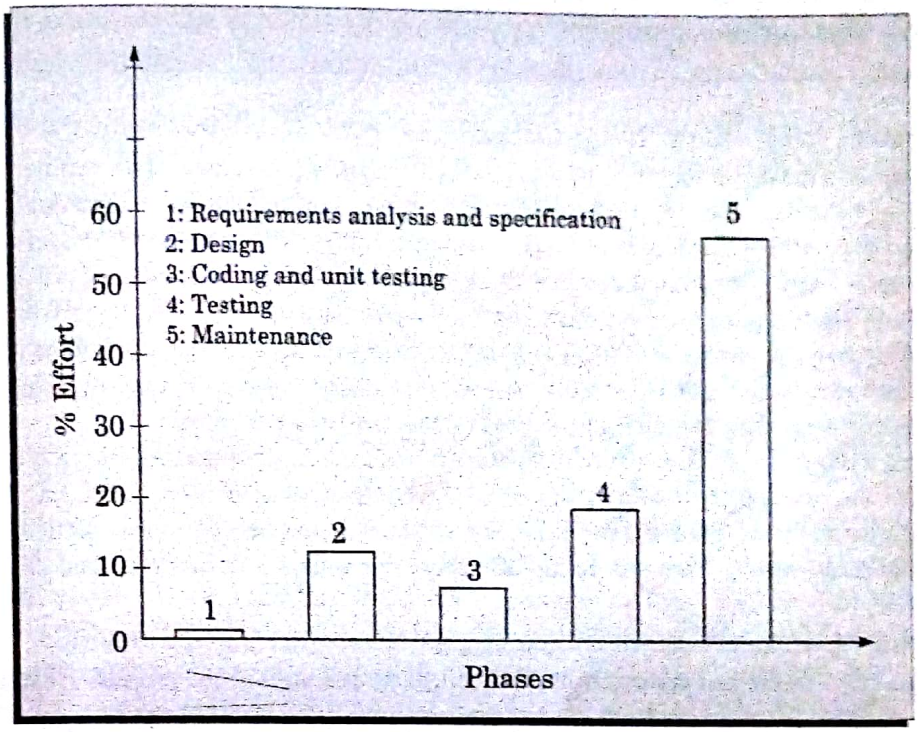

Figure 2.2: Relative effort distribution among different phases of a typical product.

Feasibility study

The main aim of the feasibility study activity is to determine whether it would be *financially* and *technically feasible* to develop the product. The feasibility study activity involves analysis of the problem and collection of all relevant information relating to the product such as the different data items which 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 behaviour of the system. These collected data are analyzed to arrive at the following:

1. **An abstract problem definition:** Only the important requirements of the customer are captured and the details of the requirements are ignored.

2. **Formulation of the different strategies for solving the problem:** All the different ways in which the problem can be solved are identified.

3. **Evaluation of the different solution strategies:** The different solution strategies are analyzed to examine their benefits and shortcomings. This analysis usually requires making approximate estimates of the resources required, cost of development, and development time required for each of the alternate solutions. These estimates are used as the basis for comparing the different solutions. Once the best solution is identified, all later phases of development

are carried out to as per this solution. In other words, we can say that during the feasibility study, very high-level decisions regarding the exact solution strategy to be adopted are made. Therefore, feasibility study is considered to be a very important stage. 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.

The following case study is an example of a feasibility study undertaken by an organization. It is intended to give you a feel of the activities and issues involved in the feasibility study phase of a typical software project.

Case study

A mining company named Galaxy Mining Company Ltd. (GMC) 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 instalments from each miner every month and deposit the same to the CSPFC (central special provident fund commissioner). The CSPFC will maintain all details regarding the SPF installments collected from the miners.

GMC 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 has realized that besides saving manpower on bookkeeping work, the software would help in speedy settlement of claim cases. GMC indicated that the amount it can afford for this software to be developed and installed is Rs. 1 million.

Adventure Software Inc. deputed their project manager to carry out the feasibility study. The project manager discussed with the top managers of GMC 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 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 analyzed the database functionalities required, the user-interface issues, and the software handling communication with the mine sites. He arrived at a cost to develop from this analysis. 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 his solution with the GMC management and found that the 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 as follows:

1. Requirements gathering and analysis: This activity consists of first gathering the requirements and then analyzing the gathered requirements. The goal of the requirements gathering activity is to collect all relevant information regarding the product to be developed from the customer with a view to clearly understand the customer requirements. Once the requirements have been gathered, the analysis activity is taken up. The goal of the requirements analysis activity is to weed out the incompleteness and inconsistencies in these requirements. Note that an *inconsistent* requirement is one where some part of the requirement contradicts with some other part. On the other hand, an *incomplete* requirement is one where some parts of the requirement may have been omitted inadvertently.

2. Requirements specification: The customer requirements identified during the requirements gathering and analysis activity are organized into a Software Requirements Specification (SRS) document. The three most important contents of this document are the functional requirements, the non-functional requirements, and the goals of implementation. Functional requirements describe the functions to be supported by the system. Each function can be characterized by the input data, the processing required on the input data, and the output data to be produced. The non-functional requirements identify the performance requirements, the required standards to be followed, etc.

The SRS document is written using end-user terminology. This makes the SRS document understandable by the customer. After all, it is important that the SRS document be reviewed and approved by the customer. 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 developer 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 shall examine the requirements analysis activity and various issues involved in developing a good SRS document in 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 being used at present. These are the traditional design approach and the object-oriented design approach. 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.

1. Traditional design approach: The traditional design approach is currently being used by many software development houses. The traditional design technique is based on the data flow-oriented design approach. While using this technique the design phase consists of two important activities; first a *structured analysis* of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a *structured design* activity. During structured design, the results of structured analysis are transformed into the software design.

Structured analysis involves preparing a detailed analysis of the different functions to be supported by the system and identification of the data flow among the functions. Each function

required by the user is studied carefully and then recursively decomposed into various sub-functions. In addition to identifying the various functions (also known as processes) in the system, the data flow among the functions are also identified. Data flow diagrams (DFDs) are used to perform structured analysis and to document the results. The DFD technique is discussed in Chapter 6. Structured analysis usually restricts itself to the *what needs to be done* aspects of the problem and carefully avoids discussing the 'how to do it' aspects. During structured analysis, the functional requirements specified in the SRS document are decomposed into subfunctions and the data-flow among these subfunctions is analyzed and represented diagrammatically in the form of DFDs.

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 detailed design, internals of the individual modules are designed in greater detail, e.g., the data structures and algorithms of the modules are designed and documented. Several well-known methodologies are available for working out the architectural and low-level designs.

2. Object-Oriented design approach: Object-oriented design (OOD) is a relatively new technique. 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 has several benefits such as lower development time and effort, and better maintainability of the product. 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 of software development is to translate the software design into source code. 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 tested. To enable the engineers to write good quality programs, every software development organization normally formulates its own coding standards that suits itself. A coding standard addresses issues such as the standard ways of laying out the program codes, the template for laying out the function and module headers, commenting guidelines, variable and function naming conventions, the maximum number of source lines permitted in each module, etc.

After coding is complete, each module is unit tested. Unit testing involves testing each module in isolation from other modules, then debugging, and documenting it. The main objective of *unit testing* is to determine the correct working of the individual modules during unit testing. During unit testing, each module is tested in isolation as this is the most efficient way to debug the errors identified at this stage (can you guess the reason?). Another reason behind testing a module in isolation is that the modules with which a module under unit test needs to interfaces may not be ready. Unit testing involves a precise definition of the test cases, testing criteria, and management of test cases. We shall discuss the various coding and unit testing techniques in Chapter 10.

Integration and system testing

Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the different modules are integrated in a planned manner. The modules making up a software product 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, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities as follows:

1. **α-testing:** α testing is the system testing performed by the development team
2. **β-testing:** This is the system testing performed by a friendly set of customers
3. **Acceptance testing:** This is the system testing performed by the customer himself after the product delivery to determine whether to accept the delivered product or to reject it.

System testing is normally carried out in a planned manner according to a **system test plan** document. The system test plan identifies all testing-related activities that must be performed and specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case. Immediately after the requirements specification phase a **system test plan** can be prepared which documents the plan for system testing. It is possible to prepare the system test plan just after the requirements specification since it can be prepared solely based on the SRS document. The results of integration and system testing are documented in the form of a **test report**. The test report summarizes the outcome of all the testing that were carried out during this phase. We discuss the details of various integration and system testing techniques in Chapter 13.

Maintenance

Maintenance of a typical software product requires much more effort than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in 40:60 ratio. The proportion of effort on maintenance can be higher for long lasting software such as operating systems, certain product manufacturing software, etc. On the other hand, the proportion of maintenance effort can be low for software that are used for just a couple of years such as a software developed for a certain business application and the business itself gets obsolete. Maintenance involves performing any one or more of the following three kinds of activities:

1. **Corrective maintenance:** This type of maintenance involves correcting errors that were not discovered during the product development phase.

2. **Perfective maintenance:** This type of maintenance involves improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements.

3. 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.

Details of various maintenance activities are discussed in Chapter 13.

2.2.2 Shortcomings of the Classical Waterfall Model

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

1. The classical waterfall model considers the transition between two phases to be similar to a waterfall. That is, once a phase is complete, the various activities during the phase are assumed to be flawlessly done and there is no scope for rework at a later time.

> The classical waterfall model is an idealistic one since it assumes that no error is ever committed by the engineers during any of the life cycle phases, and therefore, leaves no scope for error correction.

However, in practical development environments, the developers do commit a large number of errors in almost every phase of the life cycle. The cause for defects can be many—oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc. 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, the engineers need to go back to the phase where the defect had occurred and redo some of the work done during that phase, and also the work of later phases affected by the rework. Therefore, in any practical software development work, it is not possible to strictly follow the classical waterfall.

2. This model assumes that all requirements are defined correctly at the beginning of the project, and on basis of that, the development work starts. However, that is seldom the case in real projects. The customers keep on changing the requirements as their development proceeds. Thus, it becomes difficult to accommodate later requirements change requests made by the customer.
3. This model assumes, that all the phases are sequential. However, that is rarely the case. For example, for efficient utilization of manpower, in a company the members assigned the testing work, may start their work immediately after the requirements specification to design the system test cases. Therefore, we can say that the design and testing phases overlap. Consequently, it is safe to say that in a practical software development scenario the different phases might overlap, rather than having a precise point in time at which one phase stops and the other starts.

2.3 ITERATIVE WATERFALL MODEL

We have seen in the previous section that in a practical software development work, it is not possible to strictly follow the classical waterfall model. We branded the classical waterfall model as an idealistic model. In this context, we can view the iterative waterfall model as making necessary changes to the classical waterfall model so that it becomes applicable to practical software development projects. Essentially, the main change to the classical waterfall model is in the form of providing feedback paths from every phase to its preceding phases as shown in Figure 2.3. The feedback paths allow for correction of the errors committed during a phase, as and when these are detected in a later phase. For example, if during testing a design error is identified, then then the feedback path allows the design to be reworked and and the changes to be reflected in the design documents. However, observe that there is no feedback path to the feasibility stage. This means that the feasibility study errors cannot be corrected.

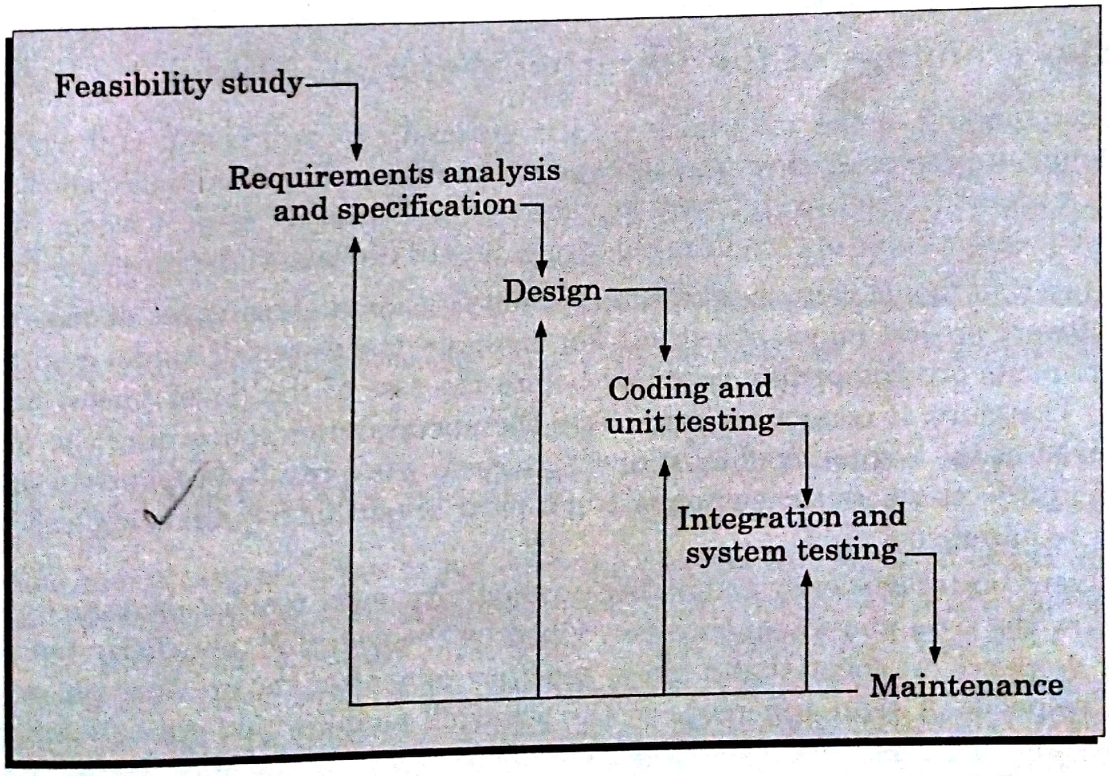

Figure 2.3: Iterative waterfall model.

2.3.2 Shortcomings of the Iterative Waterfall Model

A good understanding of the waterfall model is necessary to appreciate and understand the other development processes. However, the waterfall model suffers from many shortfalls. These shortcomings are addressed to different extents by the other life cycle models that we shall discuss subsequently. Some of the glaring shortcomings of the waterfall model are the following:

1. The waterfall model cannot satisfactorily handle the different types of risks that a real-life software project may suffer from. For example, the waterfall model assumes that the requirements are completely specified before the rest of the development activities can start. Therefore, it cannot accommodate the uncertainties concerning the requirements that exist at the beginning of most of the projects. As a result, it cannot be satisfactorily used in projects where the customer is not clear about his requirements and can provide rough requirements only.
2. To achieve better efficiency and higher productivity, most real life projects find it 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. A rigid adherence to the waterfall model would create *blocking states* in the system. That is, once a developer completes his work, he idles waiting for the phase to get over. Since the work required to be done in a phase is distributed among several team members, some members may complete their work earlier than other members

2.4 PROTOTYPING MODEL

The prototyping model requires that before carrying out the development of the actual software, a working *prototype* of the system should be built. A prototype is a toy implementation of the system. A prototype is usually built using several short cuts. The short cuts might involve using inefficient, inaccurate, or dummy functions. The short cut implementation of a function, for example, may produce the desired results by using a table look-up rather than by performing the actual computations. A prototype usually turns out to be a very crude version of the actual system, possibly exhibiting limited functional capabilities, low reliability, and inefficient performance as compared to the actual software.

by Brooks [1975].

The prototyping model of software development is shown in Figure 2.5. As shown Figure 2.5, the first phase is prototype development to control various risks. This is follow by an iterative development cycle. In this model, prototyping starts with an initial requ ments gathering phase. A quick design is carried out and a prototype is built. The develop prototype is submitted to the customer for his evaluation. Based on the customer feedba the requirements are refined and the prototype is suitably modified. This cycle of obta ing customer feedback and modifying the prototype continues till the customer approves t prototype. Once the customer approves the prototype, the actual system is developed usi the iterative waterfall approach. In spite of the availability of a working prototype, the Sp document is required to be developed for carrying out traceability analysis, verification, a test case design during later phases. However, especially for developing GUI part using t prototyping model, the requirements analysis and specification phase would become redunda since the working prototype that has been approved by the customer can serve as an animat requirements specification.

The code for the prototype is usually thrown away. However, the experience gathered fro developing the prototype helps a great deal in developing the actual system. Therefore, eve though the construction of a working prototype might involve additional cost, for system with unclear customer requirements and for systems with unresolved technical issues, th overall development cost might turn out to be lower in the prototype model than that of a equivalent system developed using the iterative waterfall model. By constructing the prototyp and submitting it for user evaluation, many customer requirements get properly defined an technical issues get resolved by experimenting with the prototype. This minimizes the chang requests from the customer and the associated redesign costs.

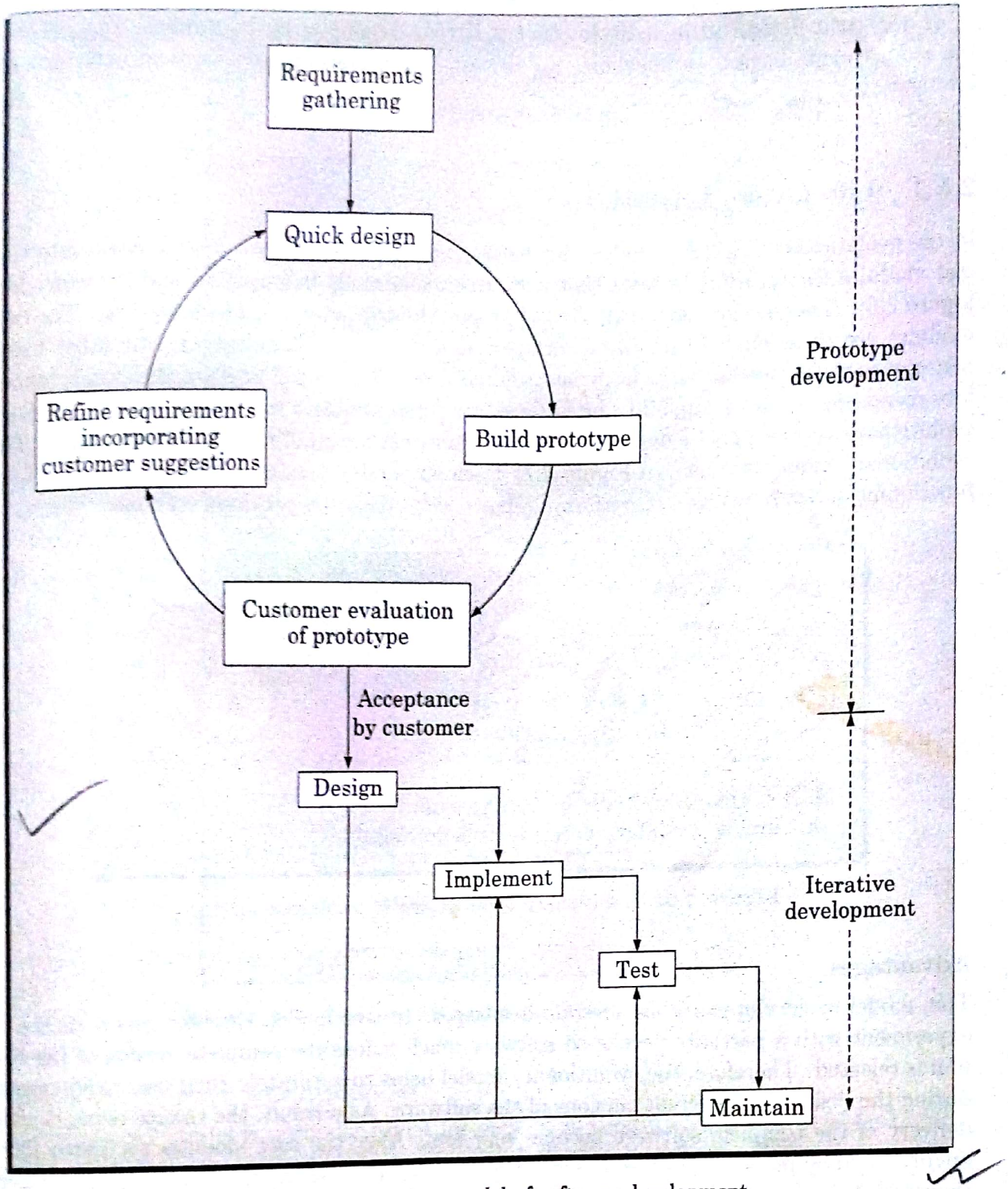

Figure 2.5: Prototyping model of software development.

2.5 EVOLUTIONARY MODEL

This life cycle model is also referred to as the **successive versions** model and sometimes as the **incremental model**. In this life cycle model, first a simple working system is built, which subsequently undergoes many functionality improvements and additions until the desired system is realized. The evolutionary software development process is therefore sometimes referred

to as **design a little, build a little, test a little, deploy a little model**. That is, once the requirements have been specified, the design, build, test, and deployment activities are interleaved.

2.5.1 Life Cycle Activities

In the evolutionary life cycle model, the software requirement is first broken down into several modules (or functional units) that can be incrementally constructed and delivered (see Figure 2.6). The development team first develops the core modules of the system. The core modules are those that do not need services from the other modules. On the other hand, non-core modules need services from the core modules. This initial product skeleton is refined into increasing levels of capability by adding new functionalities in successive versions. Each evolutionary version may be developed using an iterative waterfall model of development. The evolutionary model is shown in Figure 2.7. Each successive version of the product is a fully functioning software capable of performing more work than the previous versions.

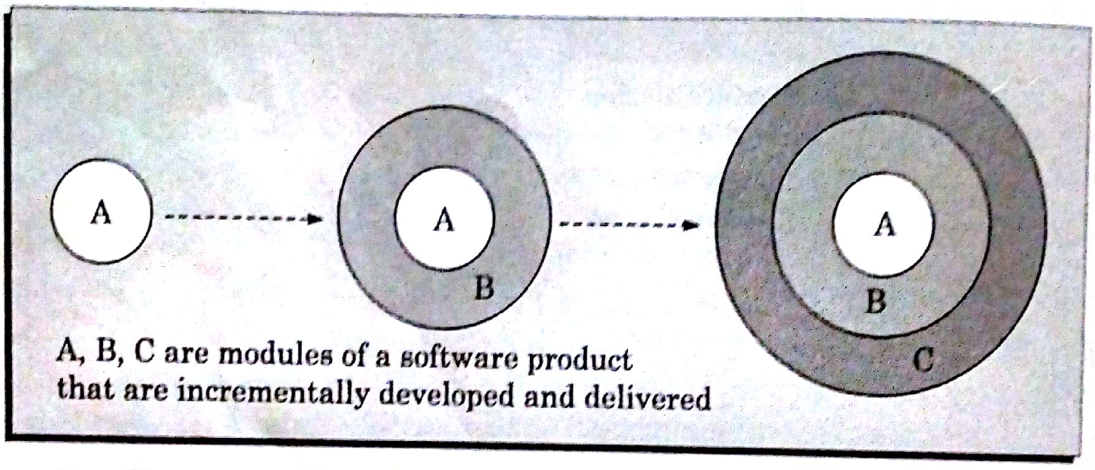

Figure 2.6: Evolutionary development of a software product.

Advantages

This model of development has several advantages. In this model, the user gets a chance to experiment with a partially developed software much before the complete version of the system is released. Therefore, the evolutionary model helps to accurately elicit user requirements during the delivery of different versions of the software. As a result, the change requests after delivery of the complete software become very less. Also, the core modules get tested thoroughly, thereby reducing chances of errors in the core modules of the final product. Further, this model obviates the need to commit large resources in one go for development of the system.

Disadvantages

The main disadvantage of the successive versions model is that for most practical problems it is difficult to divide the problem into several versions that would be acceptable to the customer and which can be incrementally implemented and delivered.

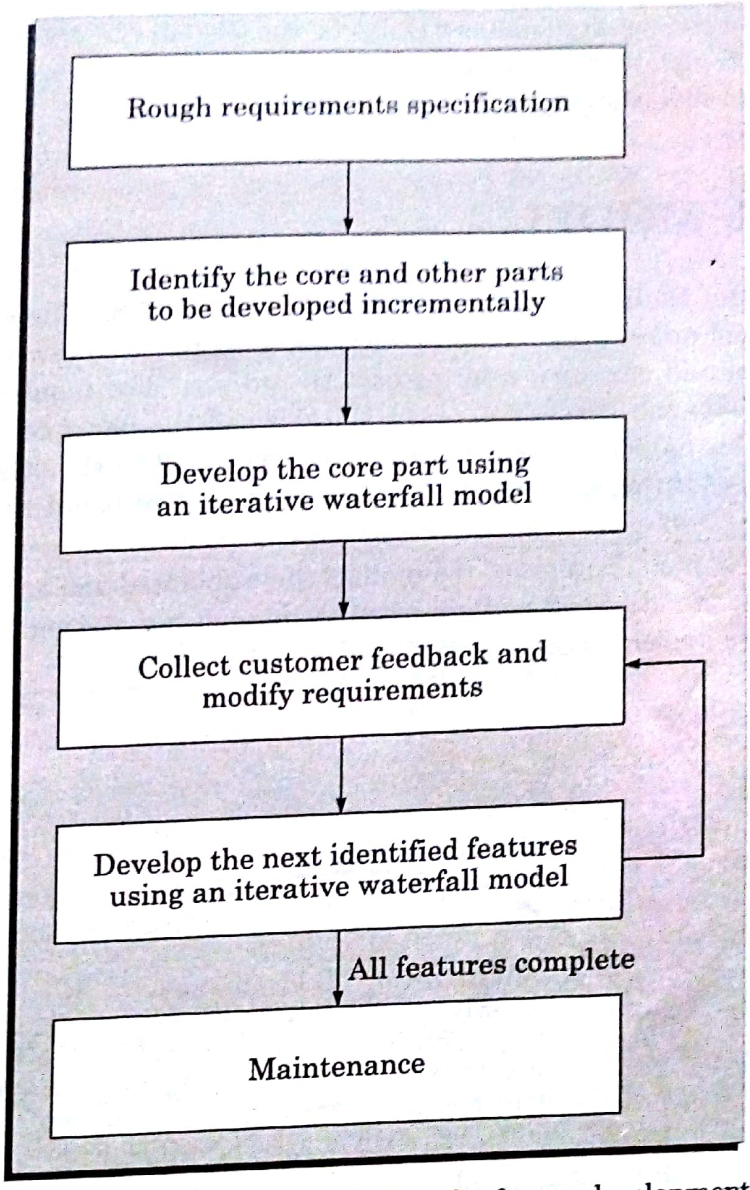

Figure 2.7: Evolutionary model of software development.

Types of projects for which suitable

The evolutionary model is normally useful for very large products, where it is easier to find modules for incremental implementation. Often evolutionary model is used when the customer prefers to receive the product in increments so that he can start using the different features as and when they are developed rather than waiting all the time for the full product to be developed and delivered. Another important category of projects for which the evolutionary model is suitable is the following.

The evolutionary model is a very natural model to use in object-oriented software development projects.

Because in object-oriented development projects, the system can easily be partitioned into stand-alone units in terms of the objects. Also, objects are more or less self-contained units that can be developed independently.

2.6 SPIRAL MODEL

The spiral model of software development is shown in Figure 2.8. The diagrammatic representation of this model appears like a spiral with many loops. 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 2.8 is just an example. Each loop of the spiral is called a phase of the software process. It can be seen that this model is much more flexible compared to the other models, since the exact number of phases through which the product is developed is not fixed.

Over each loop, one or more features of the product are elaborated and analyzed and the risks at that point of time are identified and are resolved through prototyping. Based on this, the identified features are implemented.

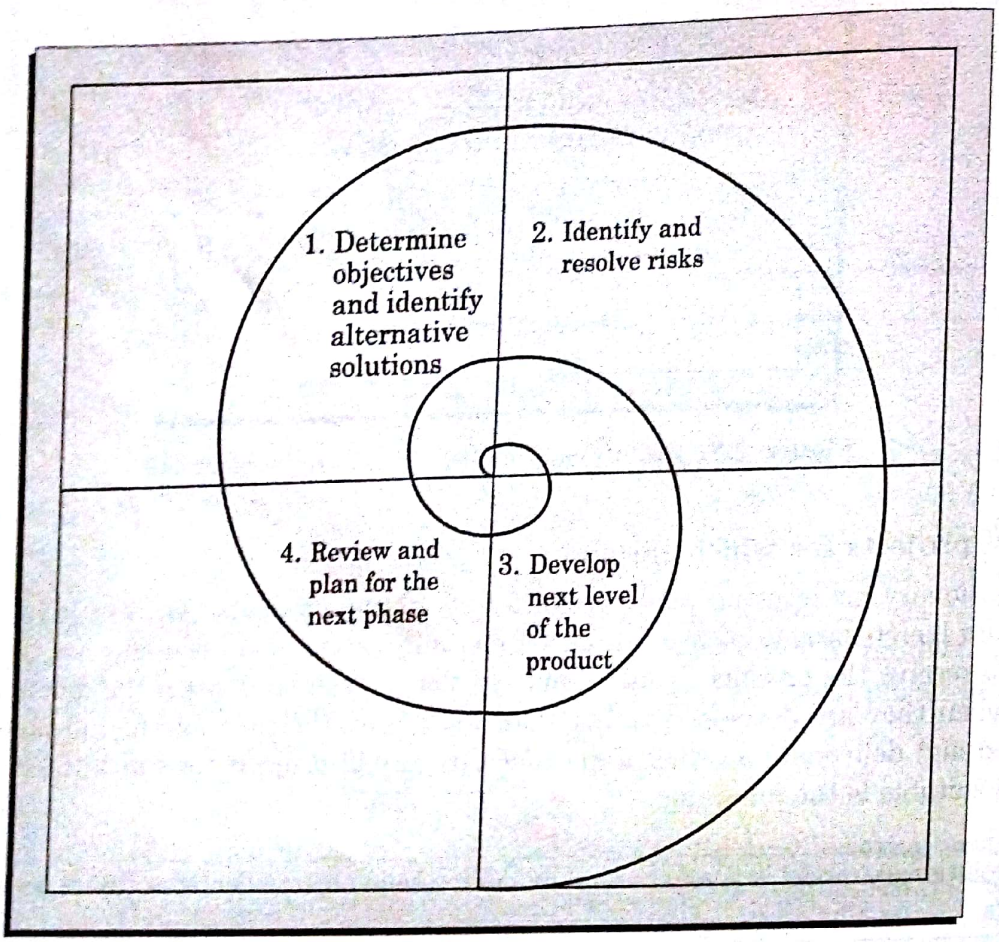

Figure 2.8: Spiral model of software development.