UNIT – I

Introduction to Software Engineering

Introduction:

Virtually all countries now depends on complex computer based systems. National infrastructure and utilities relay on computer based systems; and most electrical product includes a computer and controlling software. Therefore producing and maintaining software cost effectively is essential for functioning of national and international economies.

Software engineering is an engineering discipline whose focus is the cost effective development of high quality software system. The nation of software engineering was fast proposed in 1968 at a conference held to discuss what was then called the "Software crisis". This software crisis resulted directly from the introduction of new computer hardware based on integrated circuits.

Early experience in building this system showed that informal software development was not good enough. Major projects were sometime years late. The software cost much more than poorly. Software development was on crisis. Hardware costs were tumbling whilst software cost were rising rapidly. New technique and method were needed to control the complexity inherit in large software system.

These techniques have become part of software engineering and are now widely used. However, as our ability to produce software has increased, so too has the complexity of the software of computers and communication systems and complex graphical user interface place new demands of software engineers. There was a tremendous progress since 1968 and that the development of software engineering has markedly improved our software's. We have a much better understanding of the activities involved in software development.

Now there are effective methods of software specification, designs and implementation. New notation and tools are reduce the effort required to produce large and complex systems. Now there is no single "ideal" approach to software engineering. There are different types of system and organization that uses these systems means that we need a diversity of approaches to software development. However, fundamental notation of process and system organization underlines all of these techniques and these are the essence of software engineering.

WHAT IS SOFTWARE?

Many people equate the term software with computer programs. Software is not just program but also associated documentation and configuration data that is needed to make these programs operate correctly. A software system usually consists of number of separate programs, configuration files, system documentation and user documentation. There are two fundamental types of software products.

1. Generic products: These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them.

Ex: Software for PC's such as database, word processor.

2. Customized (or bespoke) products: These are systems which are commissioned by a particular customer. A software contractor develops the software especially for that customer.

Ex: Air traffic control system.

As an important difference between these types of software is that in generic products, the organization develops the software controls the software specification is usually developed and controlled by the organisation buying the software. The software developers must work to the specification. More and more software companies are starting with a generic system and costuming it to the needs of a particular customer.

WHAT IS SOFTWARE ENGINEERING?

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stage of systems specification to maintaining the system after it has gone into use. In the above definition there are two key phrase:

1. Engineering Discipline: Engineers make things work. They apply theories, methods and tools where there are appropriate, but they use them selectively and always try to discover solutions to problem even when there are no applicable theories and methods.

2. All aspects of Software Production: Software engineering is not just concerned with the technical process of software development but also with the activities such as software project management and with the development of tools, methods, and theories to support software production.

In general, software engineers adopt a systematic and organised approach to their work, as this is often the most effective way to produce high quality software.

WHAT 'S THE DIFFERENCE BETWEEN SOFTWARE ENGINEERS AND COMPUTER SCIENCE?

Essentially, computer science is concerned with theories and methods that underline computers and software systems whereas software engineering is concerned with the practical problem of producing software, some knowledge of computer science is essential for software engineers.

All of software engineering should be underpinned by theories of computer science, but in reality this is not the case. Elegant theories of computers science cannot always be applied to real, complex problems that require software solutions.

WHAT IS THE DIFFERENCE BETWEEN SOFTWARE ENGINEERING AND SYSTEM ENGINEERING?

System engineering is concerned with all aspects of the development and evaluation of complex systems where software plays a major role. System engineering is therefore concerned with hardware development policy and process design and development as well as software engineering. System engineering is Involved in specifying the system defining its overall architecture and the integrating the different parts to create the finished systems. They are less concerned with the engineering of the system component.

WHAT IS SOFTWARE PROCESS?

A software process is the set of activities and associated result that produce a software product. There are four fundamental process activities and they are:

1. Software specification where customers and engineer define the software to be produced and the constant on its operation.
2. Software development where the software is design and programmed.
3. Software validation where the software is checked to ensure that it is what the customer requires.
4. Software evaluation where the software is modified to adopt it to changing customer and market requirements.

Different types of system need different development process. Use of an inappropriate software process may reduce the quality or the usefulness of the software product to be developed and increase the development cost.

WHAT IS A SOFTWARE PROCESS MODEL?

A software process model is a simplified description of a software process that presents one view of that process. Process may include activities that are part of the software process software products and the roles of people involved in software engineering. Types of software process model that may be produced are:

1. A work flow model: This shows the sequence of activities in the process along with their inputs outputs and dependencies. The activities in this model represent human actions.

2. A dataflow or activity model: This represents the process as a set of activities, each of which carries out some data transformation. It shows how the input to the process, such as a design. The activities here many represents transformation carried out by people or by computers.

3. A role/action model: This represents the roles of the people involved in the software process and the activities for which they are responsible.

Most software process models are based on one of three general models or generic models:

a. **Waterfall Approach:** This takes the above activities and represents them as separate process phases such as requirement, specification, and software design, implementation, testing and so on. After each stage is defined it is "signed off" and development goes on to the following stage.

b. **Interactive development:** This approach inter leaves the activities of specifications, developments, and validation. An initial system is rapidly developed from very abstract specifications. This is then refined with customer input to produce a system that he satisfied the customer needs. The system made then be delivered. Alternatively, maybe Re implemented using a most structured approach produces a more robust and maintainable systems.

c. **Component Based Software Engineers (CBSE):** This technique assume that parts of the system already exist. The system development process focuses on integrating this parts rather than developing them from scratch.

What is CASE?

The acronym CASE stands for Computer Aided Software Engineering. It covers a wide range of different types of programs that are used to support software process activities such as requirements, analysis, system Modeling, debugging and testing.

All methods now come with associated CASE technology such as editors for the Nations used in the methods, analysis modules which check the system model according to the model rules and report generators to help create system documentation. The CASE tools may also include a code generator that automatically generates source code from the system model and some process guidance for software engineers.

What are the attributes of good Software?

As well as the services that it provide software product have a number of the other associated attributes the reflect the quality of software. These attributes that reflect they are not directly concerned with what the software does. Rather than they reflect its behavior while it is executing and structure and organization of the source program and associated documentation.

The attributes are:

1. Maintainability
2. Dependability
3. Efficiency
4. Usability

1. Maintainability: software should be written in such a way that it may evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable consequence of a changing business environment.

2. Dependability: software dependability has a range of characteristics including readability security and safety. Defendable software should not cause physical or economic damage in the event of system failure.

3. Efficiency: software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing, time, memory utilization etc.

4. Usability: Software must be usual with without undue effort by the type of user home it is designed. This means that it should have an appropriate user interface and adequate documentation.

What are the key challenges facing software engineering?

Software Engineering in the 21st century faces 3 key challenges:

1. The heterogeneity challenge: Increasingly system are required to operate as distributed system across networks that include different types of computer and with different kinds of support systems. The heterogeneity challenge is the challenge of developing techniques for building dependable software that is flexible enough to copy with his heterogeneity.

2. The delivery challenge: Many traditional software engineering techniques are time consuming. Hiver businesses today must be responsive and change very rapidly. Their supporting software must change equally rapidly. The delivery challenge is the challenge of shorting delivery times for large and complex systems without compromising system quality.

3. The trust challenges: as software is in text wind with all aspects of our lives it is essential that we can trust that software. This is especially true for remote software system exist through a web page or web service interface. The trust challenge is to develop techniques that demonstrate that software can be trusted by its users.

Professional and Ethical Responsibility:

Software engineering is carried out within a legal and social Framework that limits the freedoms of Engineers. Software engineers must accept that their job involves white responsibilities then simply the application and technical skills.

They must also behave in an ethical and morally responsible way if they are to be respected as professionals. They should not use their skills and abilities to behave in dishonest way or in a way that will bring disrepute to the software engineering profession. Some of these are:

1. Confidentiality: We should normally respect the confidentiality of your employees and or clients irrespective of whether a formal confidentiality agreement have been signed.

2. Competence: We should not misrepresent our level of competence. We should not knowingly accept work that is outside our competence.

3. Intellectual property right: We should be aware of local law governing the use of intellectual property such as patent and copyright.

4. Computer misuse: you should not use our technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial to extremely serious.

What is software engineering?

A popular definition of software engineering is: "A systematic collection of good program development practices and techniques". Good program development techniques have resulted from research innovations as well as from the lessons learnt by programmers through years of programming experiences. An alternative definition of software engineering is: "An *engineering approach* to develop software". Based on these two point of views, we can define software engineering as follows:

> Software engineering discusses systematic and cost-effective techniques for software development. These techniques help develop software using an engineering approach.

Is software engineering a science or an art?

Several people hold the opinion that writing good quality programs is an art. In this context, let us examine whether software engineering is really a form of art or is it akin to other engineering disciplines. There exist several fundamental issues that set engineering disciplines such as software engineering and civil engineering apart from both science and arts disciplines. Let us now examine where software engineering stands based on an investigation into these issues:

- Just as any other engineering discipline, software engineering makes heavy use of the knowledge that has accrued from the experiences of a large number of practitioners. These past experiences have been systematically organised and wherever possible

theoretical basis to the empirical observations have been provided. Whenever no reasonable theoretical justification could be provided, the past experiences have been adopted as rule of thumb. In contrast, all scientific solutions are constructed through rigorous application of provable principles.

- As is usual in all engineering disciplines, in software engineering several conflicting goals are encountered while solving a problem. In such situations, several alternate solutions are first proposed. An appropriate solution is chosen out of the candidate solutions based on various trade-offs that need to be made on account of issues of cost, maintainability, and usability. Therefore, while arriving at the final solution, several iterations and are possible.
- Engineering disciplines such as software engineering make use of only well-understood and well-documented principles. Art, on the other hand, is often based on making subjective judgment based on qualitative attributes and using ill-understood principles.

From the above, we can easily infer that software engineering is in many ways similar to other engineering disciplines such as civil engineering or electronics engineering.

Evolution of an Art into an Engineering Discipline:-

Software engineering principles have evolved over the last sixty years with contributions from numerous researchers and software professionals. Over the years, it has emerged from a pure art to a craft, and finally to an engineering discipline. The early programmers used an *ad hoc* programming style. This style of program development is now variously being referred to as e*xploratory*, *build and fix*, and *code and fix* styles.

In a build and fix style, a program is quickly developed without making any specification, plan, or design. The different imperfections that are subsequently noticed are fixed.

The exploratory programming style is an informal style in the sense that there are no set rules or recommendations that a programmer has to adhere to—every programmer himself evolves his own software development techniques solely guided by his own intuition, experience, whims, and fancies. The exploratory style comes naturally to all first time programmers. Later in this chapter we point out that except for trivial problems, the exploratory style usually yields poor quality and unmaintainable code and also makes program development very expensive as well as time-consuming.

As we have already pointed out, the build and fix style was widely adopted by the programmers in the early years of computing history. We can consider the exploratory program development style as an art—since this style, as is the case with any art, is mostly guided by intuition. There are many stories about programmers in the past who were like proficient artists and could write good programs using an essentially build and fix model and some esoteric knowledge. The bad programmers were left to wonder how some programmers could effortlessly write elegant and correct programs each time. In contrast, the programmers working in modern software industry rarely make use of any esoteric knowledge and develop software by applying some well-understood principles.

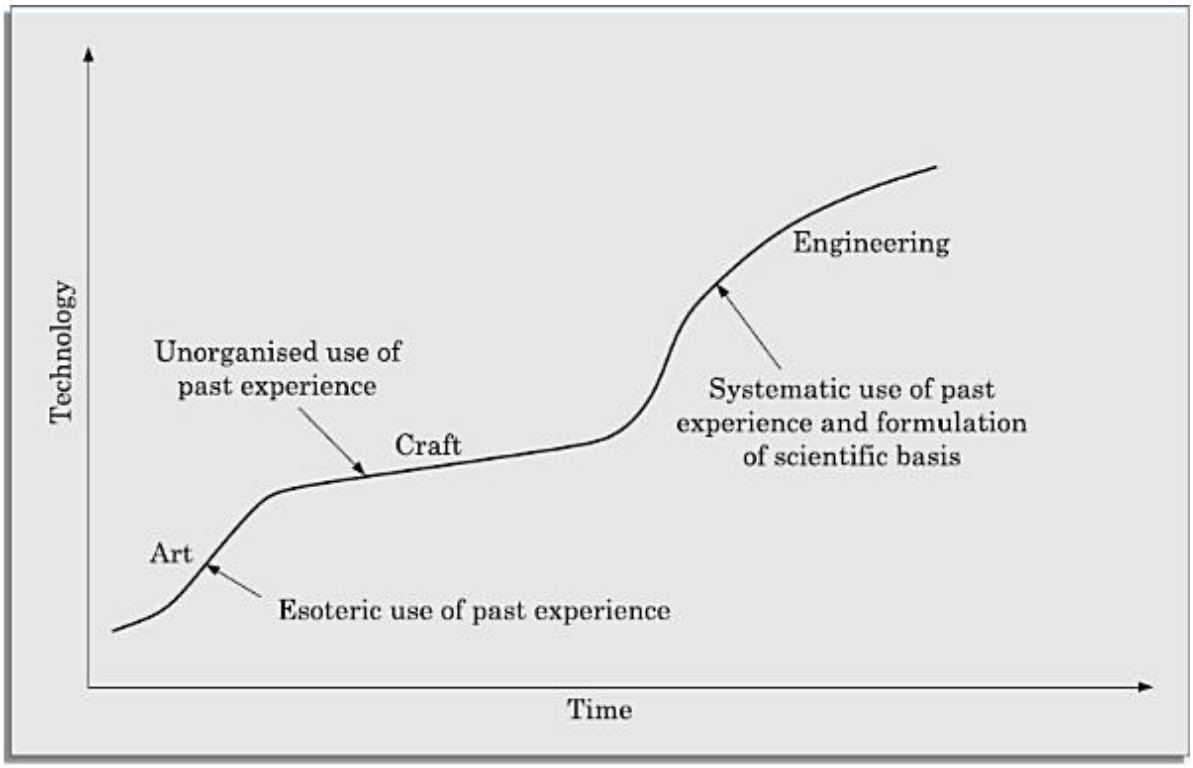
Evolution Pattern for Engineering Disciplines:-

If we analyze the evolution of the software development styles over the last sixty years, we can easily notice that it has evolved from an esoteric art form to a craft form, and then has slowly emerged as an engineering discipline. As a matter of fact, this pattern of evolution is not very different from that seen in other engineering disciplines. Irrespective of whether it is iron making, paper making, software development, or building construction; evolution of technology has followed strikingly similar patterns. This pattern of technology development has schematically been shown in below Figure. It can be seen from Figure that every technology in the initial years starts as a form of art. Over time, it graduates to a craft and finally emerges as an engineering discipline. Let us illustrate this fact using an example. Consider the evolution of the iron making technology. In ancient times, only a few people knew how to make iron. Those who knew iron making, kept it a closely-guarded secret.

This esoteric knowledge got transferred from generation to generation as a family secret. Slowly, over time technology graduated from an art to a craft form where tradesmen shared their knowledge with their apprentices and the knowledge pool continued to grow. Much later, through a systematic organisation and documentation of knowledge, and incorporation of scientific basis, modern steel making technology emerged. The story of the evolution of the software engineering discipline is not much different.

As we have already pointed out, in the early days of programming, there were good programmers and bad programmers. The good programmers knew certain principles (or tricks) that helped them write good programs, which they seldom shared with the bad programmers.

Program writing in later years was akin to a craft. Over the next several years, all good principles (or tricks) that were organized into a body of knowledge that forms the discipline of software engineering.



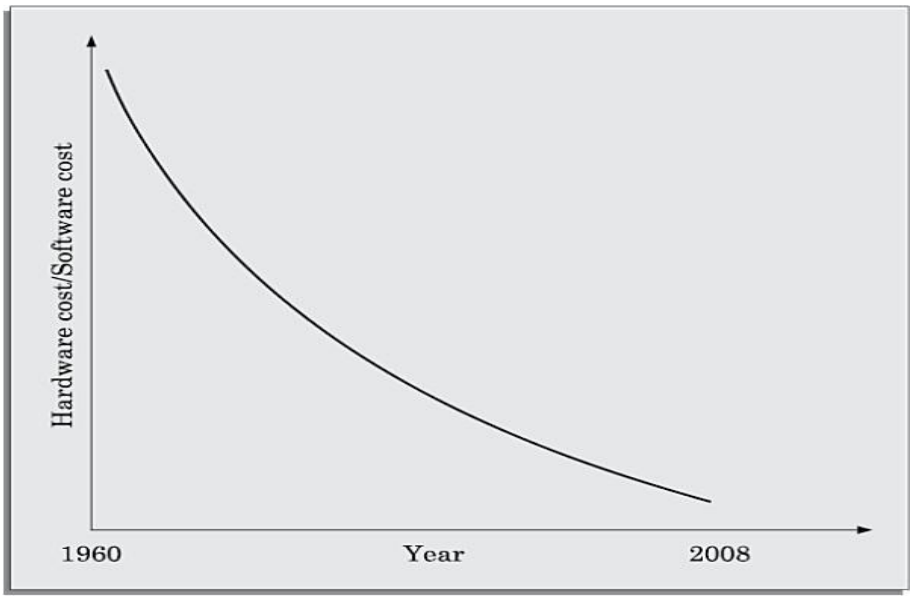
Evolution of technology with time.

Software engineering principles are now being widely used in industry, and new principles are still continuing to emerge at a very rapid rate—making this discipline highly dynamic. In spite of its wide acceptance, critics point out that many of the methodologies and guidelines provided by the software engineering discipline lack scientific basis, are subjective, and often inadequate. Yet, there is no denying the fact that adopting software engineering techniques facilitates development of high quality software in a cost-effective and timely manner. Software engineering practices have proven to be indispensable to the development of large software products—though exploratory styles are often used successfully to develop small programs such as those written by students as classroom assignments.

A Solution to the Software Crisis:-

At present, software engineering appears to be among the few options that are available to tackle the present software crisis. But, what exactly is the present software crisis? What are its symptoms, causes, and possible solutions? To understand the present software crisis, consider the following facts. The expenses that organisations all over the world are incurring on software purchases as compared to the expenses incurred on hardware purchases have been showing an worrying trend over the years (see Figure). As can be seen in the figure, organisations are spending increasingly larger portions of their budget on software as compared to that on

hardware. Among all the symptoms of the present software crisis, the trend of increasing software costs is probably the most vexing.



Relative changes of hardware and software costs over time.

Not only are the software products becoming progressively more expensive than hardware, but they also present a host of other problems to the customers—software products are difficult to alter, debug, and enhance; use resources non-optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late.

At present, many organisations are actually spending much more on software than on hardware. If this trend continues, we might soon have a rather amusing scenario. N o t long ago, when you bought any hardware product, the essential software that ran on it came free with it. But, unless some sort of revolution happens, in not very distant future, hardware prices would become insignificant compared to software prices—when you buy any software product the hardware on which the software runs would come free with the software!!!

The symptoms of software crisis are not hard to observe. But, what are the factors that have contributed to the present software crisis? Apparently, there are many factors, the important ones being—rapidly increasing problem size, lack of adequate training in software engineering techniques, increasing skill shortage, and low productivity improvements. What is the remedy? It is believed that a satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the developers, coupled with further advancements to the software engineering discipline itself.

With this brief discussion on the evolution and impact of the discipline of software engineering, we now examine some basic concepts pertaining to the different types of software development projects that are undertaken by software companies.

SOFTWARE DEVELOPMENT PROJECTS

Before discussing about the various types of development projects that are being undertaken by software development companies, let us first understand the important ways in which professional software differs from toy software such as those written by a student in his first programming assignment.

Programs *versus* Products

Many toy software are being developed by individuals such as students for their classroom assignments and hobbyists for their personal use. These are usually small in size and support limited functionalities. Further, the author of a program is usually the sole user of the software and himself maintains the code. These toy software therefore usually lack good user-interface and proper documentation. Besides these may have poor maintainability, efficiency, and reliability. Since these toy software do not have any supporting documents such as users' manual, maintenance manual, design document, test documents, etc., we call these toy software as *programs*.

In contrast, professional software usually have multiple users and, therefore, have good user-interface, proper users' manuals, and good documentation support. Since, a software product has a large number of users, it is systematically designed, carefully implemented, and thoroughly tested. In addition, a professionally written software usual l y consists not only of the program code but also of all associated documents such as requirements specification document, design document, test document, users' manuals, etc. A further difference is that professional software are often too large and complex to be developed by any single individual. It is usually developed by a group of developers working in a team.

A professional software is developed by a group of software developers working together in a team. It is therefore necessary for them to use some systematic development methodology. Otherwise, they would find it very difficult to interface and understand each other's work, and produce a coherent set of documents.

Even though software engineering principles are primarily intended for use in development of professional software, many results of software engineering can effectively be used for development of small programs as well. However, when developing small programs for personal use, rigid adherence to software engineering principles i s often not worthwhile. An ant can be killed using a gun, but it would be ridiculously inefficient and inappropriate. CAR Hoare [1994] observed that rigorously using software engineering principles to develop toy programs is very much like employing civil and architectural engineering principles to build sand castles for children to play.

Types of Software Development Projects

A software development company is typically structured into a large number of teams that handle various types of software development projects. These software development projects concern the development of either software product or some software service. In the following subsections, we distinguish between these two types of software development projects.

Software products

We all know of a variety of software such as Microsoft's Windows and the Office suite, Oracle DBMS, software accompanying a camcorder or a laser printer, etc. These software are available off-the-shelf for purchase and are used by a diverse range of customers. These are called *generic software products* since many users essentially use the same software. These can be purchased off-the-shelf by the customers. When a software development company wishes to develop a generic product, it first determines the features or functionalities that would be useful to a large cross section of users. Based on these, the development team draws up the product specification on its own. Of course, it may base its design discretion on feedbacks collected from a large number of users. Typically, each software product is targetted to some market segment (set of users). Many companies find it advantageous to develop *product lines* that target slightly different market segments based on variations of essentially the same software. For example, Microsoft targets desktops and laptops through its *Windows* 8 operating system, while it targets high-end mobile handsets through its *Windows mobile* operating system, and targets servers through its *Windows server* operating system.

Software services

A software service usually involves either development of a *customized software* or development of some specific part of a software in an outsourced mode. A *customized software* is developed according to the specification drawn up by one or at most a few customers. These need to be developed in a short time frame (typically a couple of months), and at the same time the development cost must be low. Usually, a developing company develops customized software by tailoring some of its existing software. For example, when an academic institution wishes to have a software that would automate its important activities such as student registration, grading, and fee collection; companies would normally develop such a software as a customised product. This means that for developing a customised software, the developing company would normally tailor one of its existing software products that it might have developed in the past for some other academic institution.

SOFTWARE LIFE CYCLE MODELS

We pointed out a few important differences between the exploratory program development style and the software engineering approach. 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 does 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 turns out to be 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 emphasize software development through a well-defined and ordered set of activities.

These activities are graphically modelled (represented) as well as textually described and 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.

A FEW BASIC CONCEPTS

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 an 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 then it is discarded.

As we have already pointed out, the life cycle of every software starts with a request for it 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 a new software having improved features and 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.

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, analyze those requirements, and formally document the requirements in the form of an SRS document.

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 ma y fit the same SDLC. An SDLC is represented graphically by drawing various stages of the life cycle and showing the transitions among the phases. This graphical model is usually accompanied by a textual description of various activities that need to be carried o u t during a phase before that phase can be considered to be complete. In simple words, we can define an SDLC as follows:

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, 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 to the 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.

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

Suppose a single programmer is developing a small program. For example, a student may be developing code for a class room 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 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 ma y 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 project failures in the past! 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, if 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.

Why document a development process?

It is not enough for an organisation to just have a well-defined development process, but t h e 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 develop only an informal understanding of the development process. An 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. Also, wherever necessary the methodologies for carrying out the respective activities are described. Without documentation, the activities and their ordering tend to be loosely defined, leading to confusion and misinterpretation by different teams in the organisation. For example, code reviews may informally and inadequately be carried out since there is no documented methodology as to how the 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 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 with any of the quality standards. 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

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.

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 certain specific software development situations to realize all other software life cycle models. For this reason, after discussing the classical and iterative waterfall models, we discuss its various extensions.

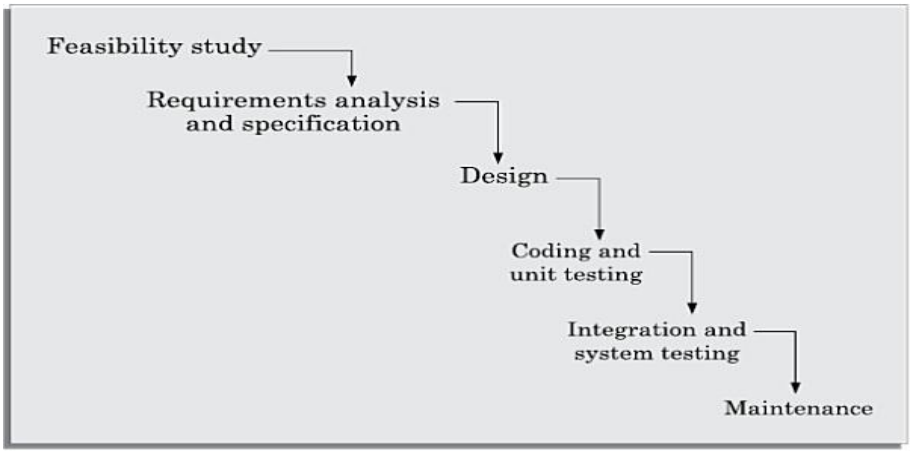
Classical Waterfall Model

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. 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.

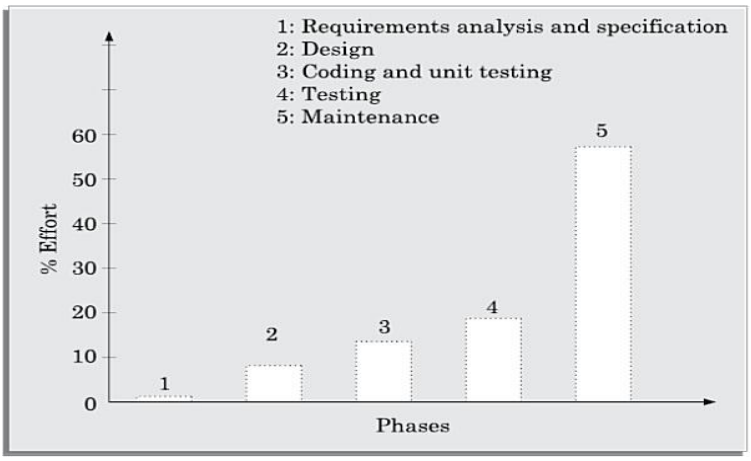
The classical waterfall model divides the life cycle into a set of phases as shown in below Figure. It can be easily observed from this figure that the diagrammatic representation of the classical waterfall model resembles a multi-level waterfall. This resemblance justifies the name of the model.



Phases of the classical waterfall model

The different phases of the classical waterfall model have been shown in above Figure. As shown in Figure, 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 signaling 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.

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 below Figure. Observe from Figure 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.



➔**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 analyzed to perform at 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 the 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 analyzed 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 summarize 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.

➔**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 subsections, 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 analyzed. 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 with 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 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.

➔**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.

Procedural design approach: The traditional design approach is in use in many software development projects at the present time. This traditional design technique is based on the data flow-oriented design approach. It consists of two important activities; first *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* 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 sub-functions and the data-flow among these sub-functions 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 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.

➔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.

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

- **-testing:** testing is the system testing performed by the development team.
- **-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.

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. 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.

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. 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 utilization of manpower, the testing team might need to design the system test cases immediately after requirements specification is complete. 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.

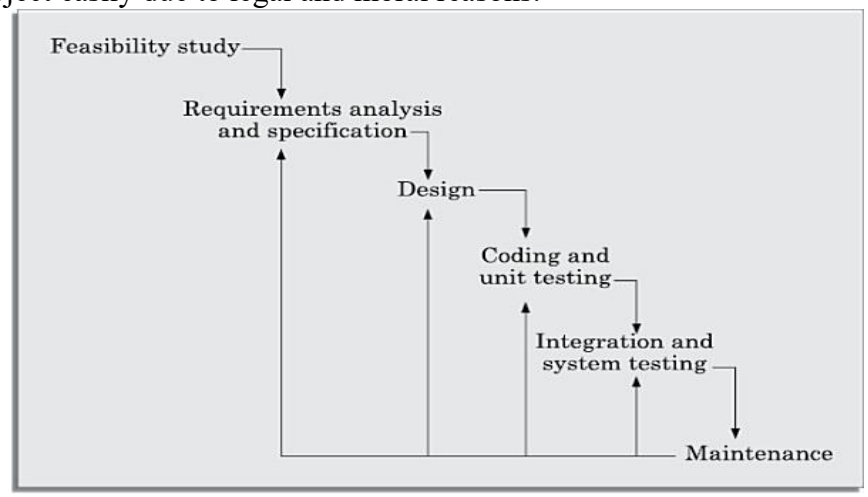
Iterative Waterfall Model

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 below Figure. 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 below Figure there is no feedback path to the feasibility stage. This is because once a team having accepted to take up a project, does not give up the project easily due to legal and moral reasons.



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*.

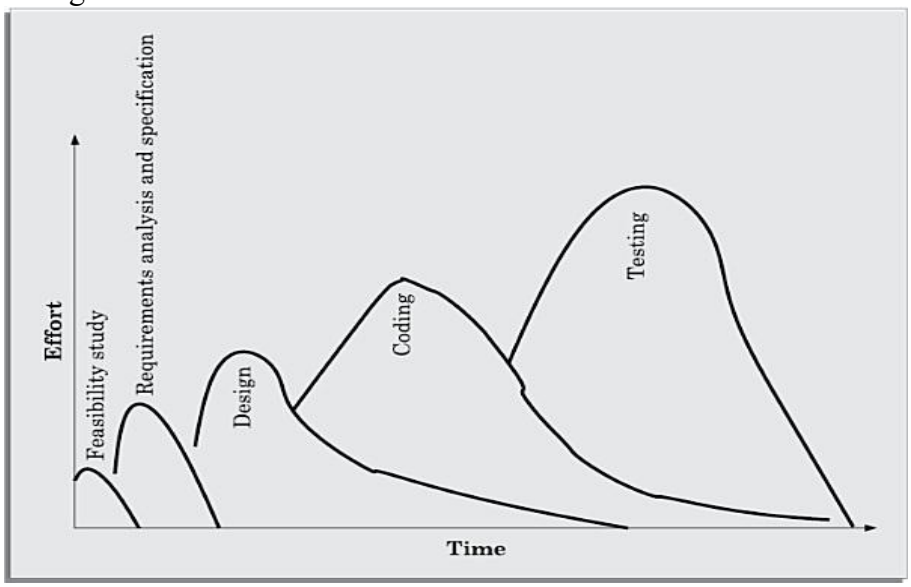
For achieving phase containment of errors, how can the developers detect almost all error that they commit in the same phase? After all, the end product of many phases are text or graphical documents, e.g. SRS document, design document, test plan document, etc. A popular technique is to rigorously review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



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

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.

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 f o r 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 would 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*. The waterfall model is

usually adapted for use in real-life projects by allowing overlapping of various phases as shown in Figure.

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 overemphasizes 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 code 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 types of projects usually involve significant reuse.

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 us e r 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 software in order to develop a good software later. Thus, the prototyping model can be deployed when development of highly optimized and efficient software is required. From the above discussions, we can conclude the following:

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.

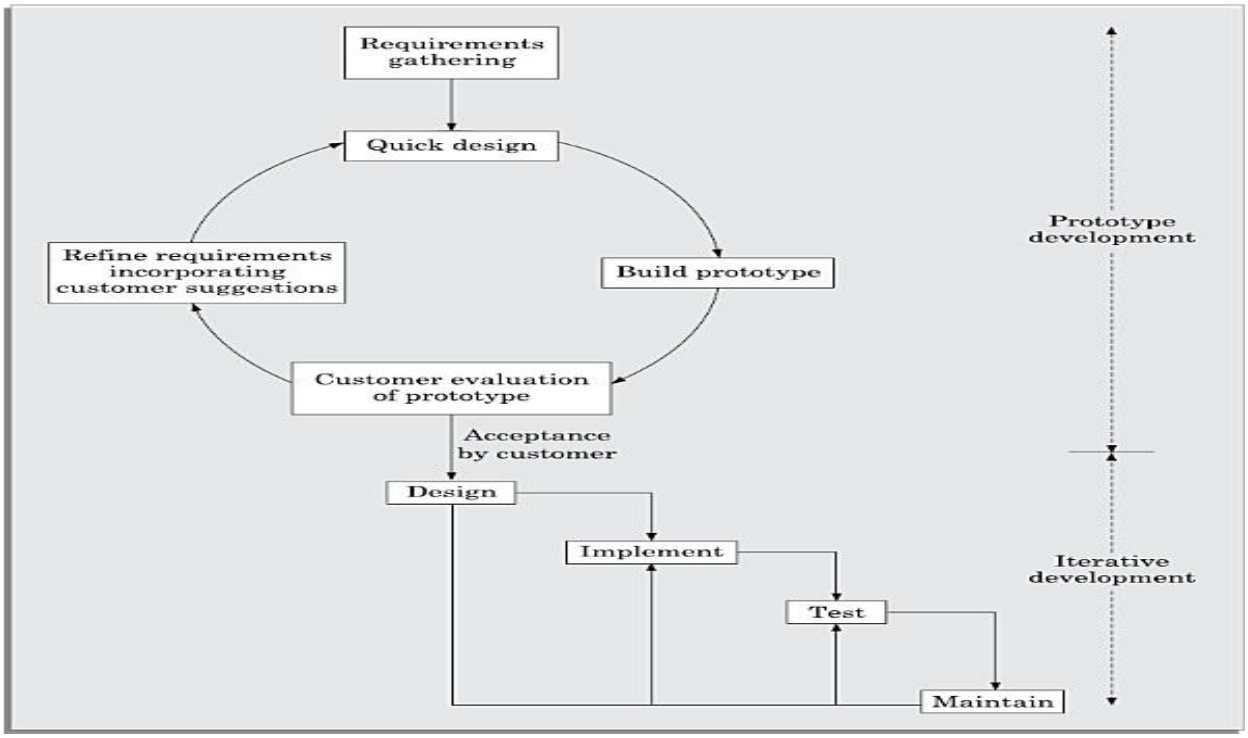
➔Life cycle activities of prototyping model
The prototyping model of software development is graphically shown in below Figure. As shown in Figure, software is developed through two major activities—prototype construction and iterative waterfall-based software development.

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 minimizes later change requests from the customer and the associated redesign costs.

➔Strengths of the prototyping model

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

➔Weaknesses of the prototyping model

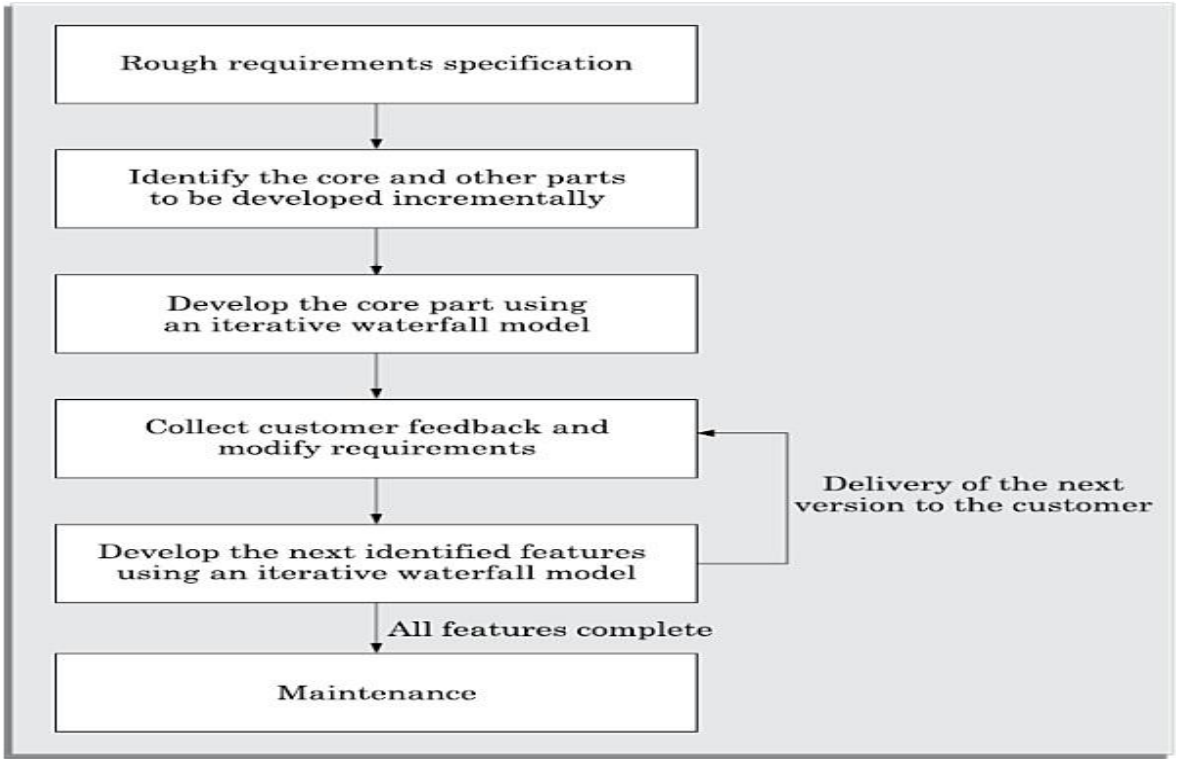
The prototype model can increase the cost of development for projects that are routine development work and do not suffer from any significant risks. Even when a project is susceptible to risks, the prototyping model is effective only for those projects for which the risks can be identified upfront before 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.

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, and 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.



➔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.

➔Applicability of the evolutionary model

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 delivered 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 projects using object-oriented development.

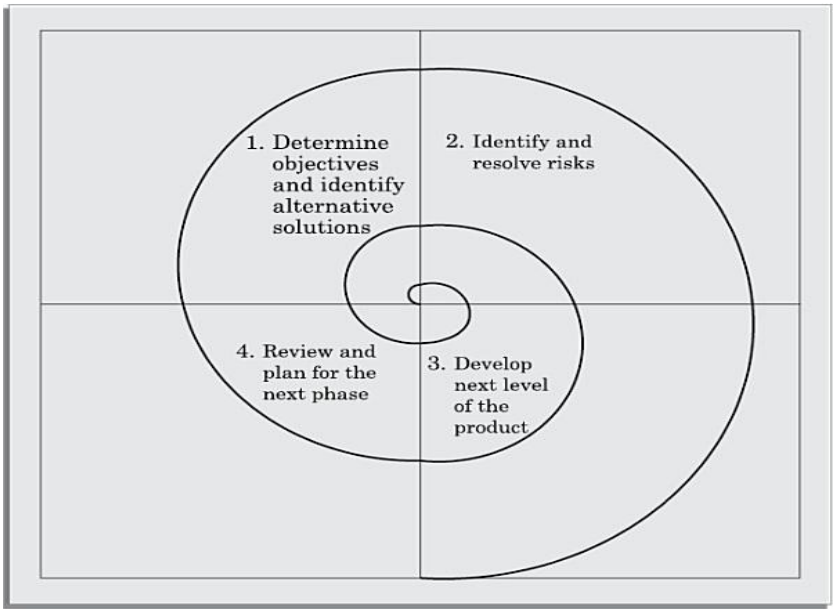
The evolutionary model is well-suited to use in object-oriented software development

Evolutionary model is appropriate for object-oriented development project, since it is easy to partition the software into stand-alone units in terms of the classes. Also, classes are more or less self-contained units that can be developed independently.

SPIRAL MODEL

This model gets its name from the appearance of its diagrammatic representation that looks 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 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 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 other models.

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 analyzed and the risks at that point of time are identified and are resolved through prototyping. Based on this, the identified features are implemented.



➔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.

Phases of the Spiral Model

Each phase in this model is split into four sectors (or quadrants) as shown in Figure. 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. 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. 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. 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 a 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.

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. In this regard, it is much more powerful than the prototyping model. Prototyping model can meaningfully 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). Selecting an Appropriate Life Cycle Model for a Project

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 favored. 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.

Advantages/pros and disadvantages/cons of the spiral model

There are a few disadvantages of the spiral model that restrict its use to a 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.

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.

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