SOFTWARE DEVELOPMENT AND COMPUTATIONAL TECHNIQUES

CPE 307

LECTURE NOTE

DEPARTMENT OF COMPUTER ENGINEERING, FACULTY OF ENGINEERING

FEDERAL UNIVERSITY OYE-EKITI

DR.ENGR. A. A. SOBOWALE Esq

A. A. SOLADOYE

PART ONE

SOFTWARE DEVELOPMENT

TABLE OF CONTENT

CHAPTER ONE

INTRODUCTION

1.1 Software Development

This is a systematic collection of good program development practices and techniques". An alternative definition is "An engineering approach to develop software. Software engineering discusses systematic and cost-effective techniques for software development. These techniques help develop software using an engineering approach.

Lots of people write programs. People in business write spreadsheet programs to simplify their jobs, scientists and engineers write programs to process their experimental data, and hobbyists write programs for their own interest and enjoyment. However, the vast majority of software development is a professional activity where software is developed for specific business purposes, for inclusion in other devices, or as software products such as information systems, CAD systems, etc. Professional software, intended for use by someone apart from its developer, is usually developed by teams rather than individuals. It is maintained and changed throughout its life

1.2 Reason for Systematic Software Development

Let us now try to figure out what exactly is meant by an engineering approach to develop software. We explain this using an analogy. Suppose you have asked a petty contractor to build a small house for you. Petty contractors are not really experts in house building. They normally carry out minor repair works and at most undertake very small building works such as the construction of boundary walls. Now faced with the task of building a complete house, your petty contractor would draw upon all his knowledge regarding house building. Yet, he may often be clueless regarding what to do. For example, he might not know the optimal proportion in which cement and sand should be mixed to realise sufficient strength for supporting the roof. In such situations, he would have to fall back upon his intuitions. He would normally succeed in his work, if the house you asked him to construct is sufficiently small. Of course, the house constructed by him may not look as good as one constructed by a professional, may lack proper planning, and display several defects and imperfections. It may even cost more and take longer to build.

Now, suppose you entrust your petty contractor to build a large 50-storeyed commercial complex for you. He might exercise prudence, and politely refuse to

undertake your request. On the other hand, he might be ambitious and agree to undertake the task. In the later case, he is sure to fail. The failure might come in several forms—the building might collapse during the construction stage itself due to his ignorance of the basic theories concerning the strengths of materials; the construction might get unduly delayed, since he may not prepare proper estimates and detailed plans regarding the types and quantities of raw materials required, the times at which these are required, etc. In short, to be successful in constructing a building of large magnitude, one needs a good understanding of various civil and architectural engineering techniques such as analysis, estimation, prototyping, planning, designing, and testing. Similar is the case with the software development projects. For sufficiently small-sized problems, one might proceed according to one's intuition and succeed; though the solution may have several imperfections, cost more, take longer to complete, etc. But, failure is almost certain, if one without a sound understanding of the software engineering principles undertakes a large-scale software development work.

1.3 Definition of SOFWARE

Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.

1.4 Attributes of good software

For a software to be considered as good software that will be well acceptable, it should possess all of these listed attributes:

i. Deliverability of the users' expected functionality and performance.
ii. Maintainability by being able to go through evolution and meet the changing customers' need and requirements
iii. Dependability: Customers should be rest minded of the smooth and efficient productivity of the software, while users are assured of the security and safety of their credentials captured or provided to the product.
iv. Usability: A good software must be easily usable by its user with standard Human-Computer Interaction protocol duly considered, so as to give the user easy accessibility and usage, even without instructor or technical or IT officer to put them through

1.5 Software as an Engineering Principle and practice.

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:

i. Just as any other engineering discipline, software engineering makes heavy use of the knowledge that has accrued from the experiences of a larges 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.

ii. 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.

iii. 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 judgement based on qualitative attributes and using ill-understood principles.

CHAPTER TWO

SOFTWARE DEVELOPMENT PROCESS

2.1. Development Process

Software development process is the process of planning and managing software development. This is a coherent set of activities that software engineer embark on in order to ensure successful delivery of suitable, well performing and acceptable software product as well maintaining it after delivery. This is the process that encompasses all the phases a product goes through from feasibility study till Changes incorporation.

2.2. Software development process activities

As earlier explained, this process consist of a number of activities that helps in successful deliver of software and its deployment for proper usage and operation. These activities are explained in the following subsections:

a) Software specification/ Requirement Engineering

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. Requirements engineering particularly critical stage of the software process as errors at this stage inevitably lead to later problems in the system design and implementation. The requirements engineering process as shown in Figure 2.1 aims to produce an agreed requirements document that specifies a system satisfying stakeholder requirements. Requirements are usually presented at two levels of detail. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification. There are four main activities in the requirements engineering process:

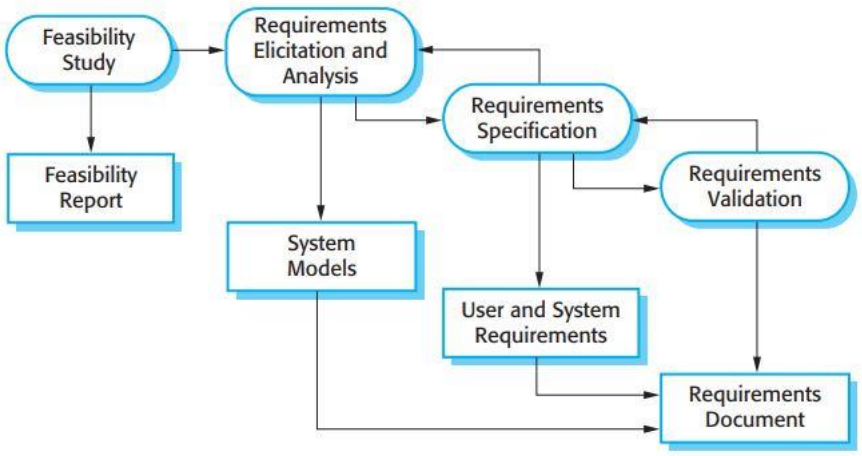

Figure 2.1: Requirement Engineering

i. *Feasibility study*: An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and if it can be developed within existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether or not to go ahead with a more detailed analysis.
ii. *Requirements elicitation and analysis*: This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. This may involve the development of one or more system models and prototypes. These help you understand the system to be specified.
iii. *Requirements specification*: Requirements specification is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document.
    - User requirements are abstract statements of the system requirements for the customer and end-user of the system.
    - System requirements are a more detailed description of the functionality to be provided.
iv. *Requirements validation*: This activity checks the requirements for realism, consistency, and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

The activities in the requirements process are not simply carried out in a strict sequence. Requirements analysis continues during definition and specification and new requirements come to light throughout the process. Therefore, the activities of analysis, definition, and specification are interleaved. In agile methods, such as extreme programming, requirements are developed incrementally according to user priorities and

b) Software development.

The implementation stage of software development is the process of converting a system specification into an executable system. It always involves processes of software design and programming but, if an incremental approach to development is used, may also involve refinement of the software specification.

A software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used. Designers do not arrive at a finished design immediately but develop the design iteratively. They add formality and detail as they develop their design with constant backtracking to correct earlier designs.

This whole component of software development phase is denoted in Figure 2.2 showing the design input, activities within the design stage and the output produced.

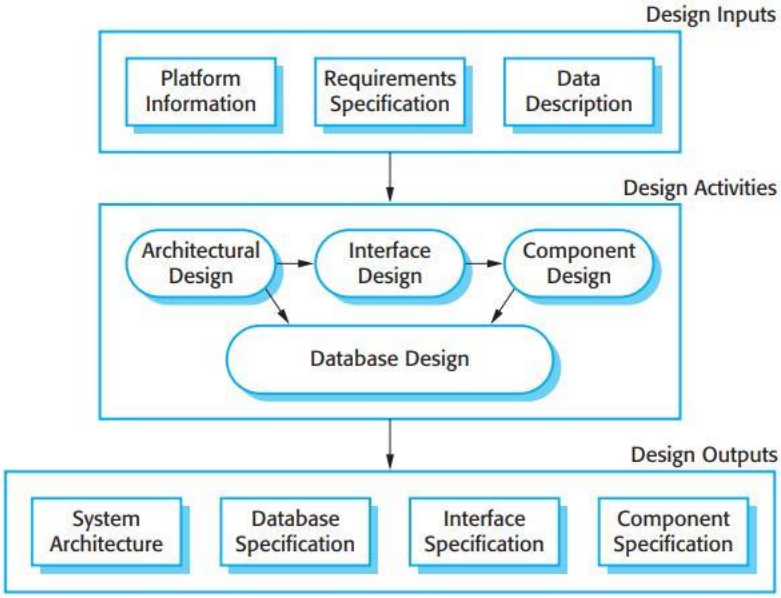

Figure 2.2: Software Design Components

The activities in the design process vary, depending on the type of system being developed. For example, real-time systems require timing design but may not include a database so there is no database design involved. Figure 2.2 shows four activities that may be part of the design process for information systems:

i. *Architectural design*: This is where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed.
ii. *Interface design*: This stage is where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.

iii. *Component design*: This is where you take each system component and design how it will operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model. The design model may be used to automatically generate an implementation.

iv. *Database design*: This is where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.

C). Software validation

Software validation or, more generally, verification and validation (V&V) is intended to show that a system both conforms to its specification and that it meets the expectations of the system customer. Program testing, where the system is executed using simulated test data, is the principal validation technique. Validation may also involve checking processes, such as inspections and reviews, at each stage of the software process from user requirements definition to program development. Because of the predominance of testing, the majority of validation costs are incurred during and after implementation.

The system is tested with the customer's data. Ideally, component defects are discovered early in the process, and interface problems are found when the system is integrated. However, as defects are discovered, the program must be debugged and this may require other stages in the testing process to be repeated. Errors in program components, say, may come to light during system testing. The process is therefore an iterative one with information being fed back from later stages to earlier parts of the process.

The stages in the testing process are:

i. *Development testing*: The components making up the system are tested by the people developing the system. Each component is tested independently, without other system components. Components may be simple entities such as functions or object classes, or may be coherent groupings of these entities. Test automation tools, such as JUnit (Massol and Husted, 2003), that can re-run component tests when new versions of the component are created, are commonly used.

ii. *System testing:* System components are integrated to create a complete system. This process is concerned with finding errors that

result from unanticipated interactions between components and component interface problems. It is also concerned with showing that the system meets its functional and non-functional requirements, and testing the emergent system properties. For large systems, this may be a multi-stage process where components are integrated to form subsystems that are individually tested before these sub-systems are themselves integrated to form the final system.

iii. *Acceptance testing*: This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data. Acceptance testing may also reveal requirements problems where the system's facilities do not really meet the user's needs or the system performance is unacceptable.

d. Software evolution

The flexibility of software systems is one of the main reasons why more and more software is being incorporated in large, complex systems. Once a decision has been made to manufacture hardware, it is very expensive to make changes to the hardware design. However, changes can be made to software at any time during or after the system development. Even extensive changes are still much cheaper than corresponding changes to system hardware.

Historically, there has always been a split between the process of software development and the process of software evolution (software maintenance). People think of software development as a creative activity in which a software system is developed from an initial concept through to a working system. However, they sometimes think of software maintenance as dull and uninteresting. Although the costs of maintenance are often several times the initial development costs, maintenance processes are sometimes considered to be less challenging than original software development.

This distinction between development and maintenance is increasingly irrelevant. Hardly any software systems are completely new systems and it makes much more sense to see development and maintenance as a continuum. Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process (Figure 2.3) where software is continually changed over its lifetime in response to changing requirements and customer needs.

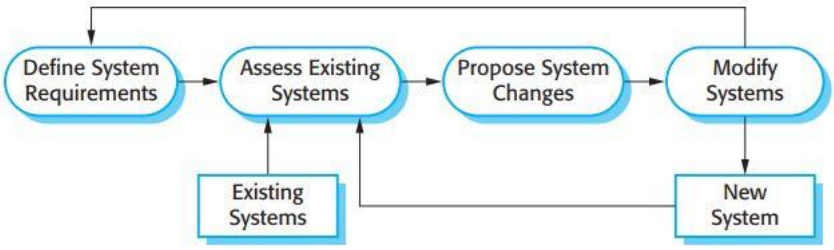

Figure 2.3: System evolution

Change is inevitable in all large software projects. The system requirements change as the business procuring the system responds to external pressures and management priorities change. As new technologies become available, new design and implementation possibilities emerge. Therefore whatever software process model is used, it is essential that it can accommodate changes to the software being developed.

Change adds to the costs of software development because it usually means that work that has been completed has to be redone. This is called rework. For example, if the relationships between the requirements in a system have been analyzed and new requirements are then identified, some or all of the requirements analysis has to be repeated. It may then be necessary to redesign the system to deliver the new requirements, change any programs that have been developed, and re-test the system. T

There are two related approaches that may be used to reduce the costs of rework:

i. Change avoidance, where the software process includes activities that can anticipate possible changes before significant rework is required. For example, a prototype system may be developed to show some key features of the system to customers. They can experiment with the prototype and refine their requirements before committing to high software production costs.
ii. Change tolerance, where the process is designed so that changes can be accommodated at relatively low cost. This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment(a small part of the system) may have to be altered to incorporate the change.

2.3. Software Process and 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.

2.4. Software Development Engineering Diversity

Software engineering is a systematic approach to the production of software that takes into account practical cost, schedule, and dependability issues, as well as the needs of software customers and producers. How this systematic approach is actually implemented varies dramatically depending on the organization developing the software, the type of software, and the people involved in the development process. There are no universal software engineering methods and techniques that are suitable for all systems and all companies. Rather, a diverse set of software engineering methods and tools has evolved over the past 50 years. Perhaps the most significant factor in determining which software engineering methods and techniques are most important is the type of application that is being developed. There are many different types of application including:

i. *Stand-alone applications*: These are application systems that run on a local computer, such as a PC. They include all necessary functioned connected to a network. Examples of such applications are office applications on a PC, CAD programs, photo manipulation software, etc.

ii. *Interactive transaction-based applications*: These are applications that execute on a remote computer and that are accessed by users from their own PCs or terminals. Obviously, these include web applications such as e-commerce applications where you can interact with a remote system to buy goods and services. This class of application also includes business systems, where a business provides access to its systems through a web browser or special-purpose client program and cloud-based services, such as mail and photo sharing. Interactive applications often incorporate a large data store that is accessed and updated in each transaction.

iii. *Embedded control systems*: These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system. Examples of embedded systems include the software in a mobile (cell) phone, software that controls anti-lock braking in a car, and software in a microwave oven to control the cooking process.

iv. *Batch processing systems*: These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs. Examples of batch systems include periodic billing systems, such as phone billing systems, and salary payment systems.

v. *Entertainment systems*: These are systems that are primarily for personal use and which are intended to entertain the user. Most of these systems are games of one kind or another. The quality of the user interaction offered is the most important distinguishing characteristic of entertainment systems.

vi. *Systems for modelling and simulation*: These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects. These are often computationally intensive and require high-performance parallel systems for execution.

vii. *Data collection systems*: These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing. The software has to interact with sensors and often is installed in a hostile environment such as inside an engine or in a remote location.

viii. *Systems of systems:* These are systems that are composed of a number of other software systems. Some of these may be generic software products,

such as a spreadsheet program. Other systems in the assembly may be specially written for that environment.

CHAPTER THREE

SOFTWARE PROCESS MODELS

3.1. PROCESS PARADIGMS

Software process model is a simplified representation of a software process. Each process model represents a process from a particular perspective, and thus provides only partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities.

These generic models are not definitive descriptions of software processes. Rather, they are abstractions of the process that can be used to explain different approaches to software development. You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.

3.2. CLASSICAL WATERFALL MODEL

The principal stages of the waterfall model directly reflect the fundamental development activities:

3.2.1 Feasibility Study:

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

i. *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.

ii. *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.

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

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

3.2.2 Requirements Analysis and Definition

The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system 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:

i. *Requirements gathering and analysis*: The goal of the requirements gathering activity is to collect all relevant information regarding the software to be developed from the customer with a view to clearly understand the requirements. For this, first requirements are gathered from the customer and then the gathered requirements are analysed. The goal of the requirements analysis activity is to weed out the incompleteness and inconsistencies in these gathered requirements. Note that an inconsistent requirement is one in which some part of the requirement contradicts with some other part. On the other hand, an incomplete

requirement is one in which some parts of the actual requirements have been omitted.

ii. *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.

3.2.3 System and Software Design

The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.

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.

i. *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 subfunctions and the

data-flow among these subfunctions is analysed 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.

ii. *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

3.2.4. Implementation and Unit Testing

During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

Testing: testing is the system testing performed by the development team. 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.

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

3.2.5. Integration and System Testing

The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

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

3.2.6. Operation and Maintenance

Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

i. *Corrective maintenance*: This type of maintenance is carried out to correct errors that were not discovered during the product development phase.
ii. *Perfective maintenance*: This type of maintenance is carried out to improve the performance of the system, or to enhance the functionalities of the system based on customer's requests.
iii. *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.

The following phase should not start until the previous phase has finished. In practice, these stages overlap and feed information to each other. During design, problems with requirements are identified. During coding, design problems are found and so on. The software process is not a simple linear model but involves feedback from one phase to another. Documents produced in each phase may then have to be modified to reflect the changes made.

Because of the costs of producing and approving documents, iterations can be costly and involve significant rework. Therefore, after a small number of iterations, it is normal to freeze parts of the development, such as the specification, and to continue with the later development stages. Problems are left for later resolution, ignored, or programmed around. This premature freezing of requirements may mean that the system won't do what the user wants. It may

also lead to badly structured systems as design problems are circumvented by implementation tricks.

During the final life cycle phase (operation and maintenance) the software is put into use. Errors and omissions in the original software requirements are discovered. Program and design errors emerge and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages.

The waterfall model is consistent with other engineering process models and documentation is produced at each phase. This makes the process visible so managers can monitor progress against the development plan. Its major problem is the inflexible partitioning of the project into distinct stages. Commitments must be made at an early stage in the process, which makes it difficult to respond to changing customer requirements.

In principle, the waterfall model should only be used when the requirements are well understood and unlikely to change radically during system development. However, the waterfall model reflects the type of process used in other engineering projects. As is easier to use a common management model for the whole project, software processes based on the waterfall model are still commonly used.

3.2.7. SHORTCOMING OF CLASSICAL WATERFALL MODEL

This types of software development models have a number of disadvantages which doesn't make it to be widely applicable to every software development projects. Some of these shortcoming are highlighted below:

i. *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.

ii. *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.

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

iv. *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.

v. *Phase overlap not supported*: For most real life projects, i t 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.

vi. *No overlapping of phases*: This model recommends that the phases be carried out sequentially—new phase can start only after the previous one completes. However, it is rarely possible to adhere to this recommendation and it leads to a large number of team members to idle for extended periods. For example, for efficient utilisation of manpower, the testing team might need to design the system test cases immediately after requirements specification is complete.

vii. *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.

viii. *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.

ix. *Heavy weight*: The waterfall model overemphasises documentation. A significant portion of the time of the developers is spent in preparing documents, and revising them as changes occur over the life cycle.

Heavy documentation though useful during maintenance and for carrying out review, is a source of team inefficiency.

x. *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.

3.3. ITERATIVE WATERFALL MODEL

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 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.

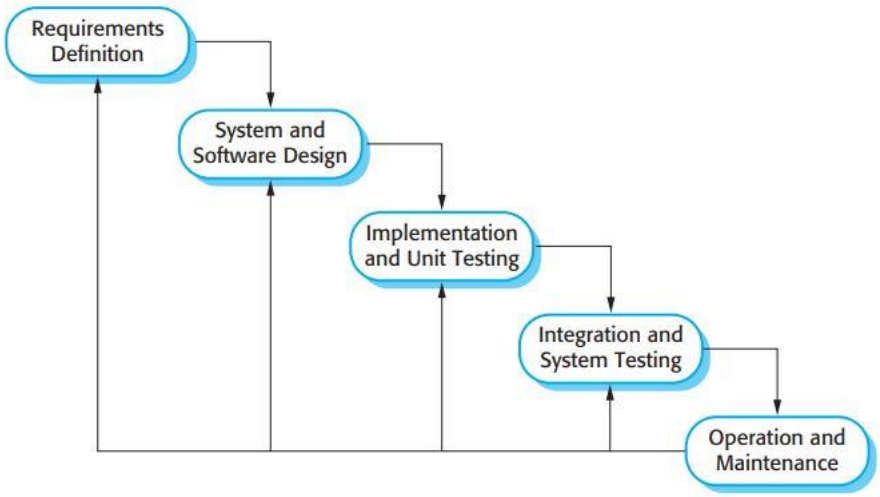

Figure 3.1: Iterative waterfall Model

Necessity of Iterative waterfall model

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 i s 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.

3.4. V MODEL

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

As shown in Figure 3.2, there are two main phases—development and validation phases. The left half of the model comprises the development phases and the right half comprises the validation phases.

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

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

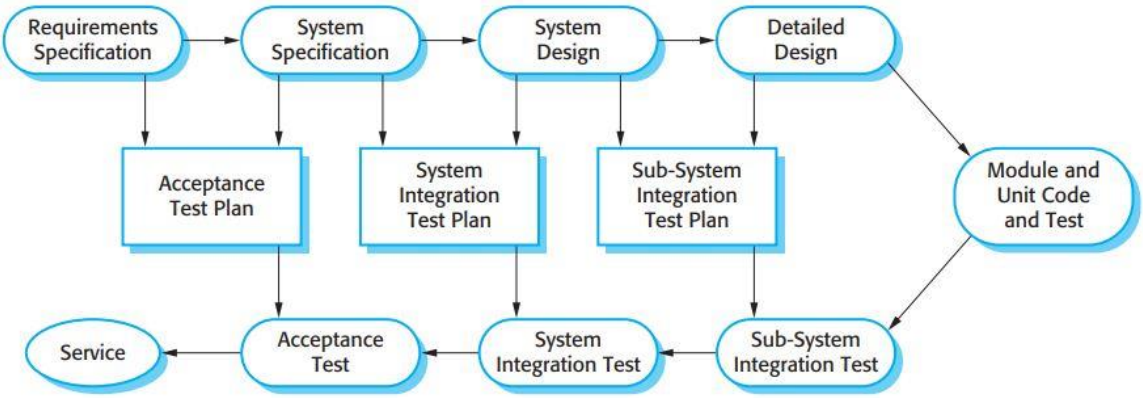

Figure 3.2: V-Model and Validation stages

We have already pointed out that the V-model can be considered to be an extension of the waterfall model. However, there are major differences between the two. As already mentioned, in contrast to the iterative waterfall model where testing activities are confined to the testing phase only, in the V-model testing activities are spread over the entire life cycle. As shown in Figure 3.2, during the requirements specification phase, the system test suite design activity takes place. During the design phase, the integration test cases are designed. During coding, the unit test cases are designed. Thus, we can say that in this model, development and validation activities proceed hand in hand.

3.4.1. Advantages of V-model

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

i. In the V-model, much of the testing activities (test case design, test planning, etc.) are carried out in parallel with the development activities. Therefore, before testing phase starts significant part of the testing activities, including test case design and test planning, is already complete. Therefore, this model usually leads to a shorter testing phase and an overall faster product development as compared to the iterative model.

ii. Since test cases are designed when the schedule pressure has not built up, the quality of the test cases are usually better.

iii. The test team is reasonably kept occupied throughout the development cycle in contrast to the waterfall model where the testers are active only during the testing phase. This leads to more efficient manpower utilisation.

iv. In the V-model, the test team is associated with the project from the beginning. Therefore they build up a good understanding of the development artifacts, and this in turn, helps them to carry out effective testing of the software. In contrast, in the waterfall model often the test team comes on board late in the development cycle, since no testing activities are carried out before the start of the implementation and testing phase.

3.4.2. Disadvantages of V-model

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

3.5. INCREMENTAL WATERFALL MODEL

Incremental development is based on the idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed. In the incremental life cycle model, the requirements of the software are first broken down into several modules or features that can be incrementally constructed and delivered.

The development team first undertakes to develop the core features of the system. The core or basic features are those that do not need to invoke any services from the other features. On the other hand, non-core features need services from the core features. Once the initial core features are developed, these are refined into increasing levels of capability by adding new functionalities in successive versions. Each incremental version is usually developed using an iterative waterfall model of development.

The simple analogy of this model is represented in Figure 3.3, whichs shows how the product moved from the initial version to the last accepted version passing through all the development Process activities concurrently.

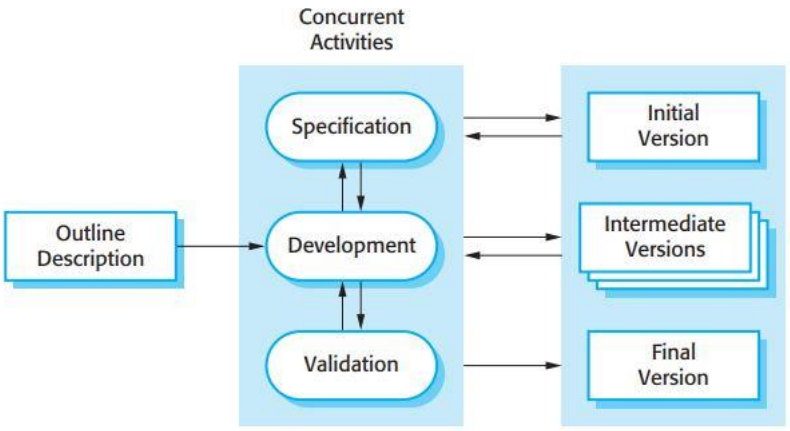

Figure 3.3: Incremental or iterative model

3.5.1. Benefits of Incremental waterfall models

Incremental development has three important benefits, compared to the waterfall model:

1. The cost of accommodating changing customer requirements is reduced. The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.

2. It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how much has been implemented. Customers find it difficult to judge progress from software design documents.

3. More rapid delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

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

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

Incremental development in some form is now the most common approach for the development of application systems. This approach can be either plan-driven, agile, or, more usually, a mixture of these approaches.

In a plan-driven approach, the system increments are identified in advance; if an agile approach is adopted, the early increments are identified but the development of later increments depends on progress and customer priorities.

3.5.2. Managerial Problem of Incremental Waterfall models

From a management perspective, the incremental approach has two problems:

1. The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.

2. System structure tends to degrade as new increments are added. Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

3.6 EVOLUTIONARY MODEL

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.

3.6.1. Advantages of Evolutionary Model

Evolutionary model as a process model for software development has some strength which makes it distinct from other model. Some of this model's advantages are highlighted as follows:

i. *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.

ii. *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.

3.6.2. Disadvantages of Evolutionary Model

As the advantages of this model was earlier stated, it also possess some weakness, making it not employable for development of some products.

I. *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.

II. *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.

3.7. RAPID APPLICATION DEVELOPMENT

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

The major goals of the RAD model are as follows:

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

3.7.1. Working of RAD

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

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

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

The decrease in development time and cost, and at the same time an increased flexibility to incorporate changes are achieved in the RAD model in two main ways—minimal use of planning and heavy reuse of any existing code through rapid prototyping. The lack of long-term and detailed planning gives the flexibility to accommodate later requirements changes. Reuse of existing code has been adopted as an important mechanism of reducing the development cost. RAD model emphasises code reuse as an important means for completing a project faster. In fact, the adopters of the RAD model were the earliest to embrace object-oriented languages and practices. Further, RAD advocates use of specialised tools

to facilitate fast creation of working prototypes. These specialised tools usually support the following features: Visual style of development. Use of reusable components.

3.7.2. Application of RAD

i. Customised software
ii. Non-critical software:
iii. Highly constrained project schedule: RAD aims to reduce development time at the expense of good documentation, performance, and reliability. Naturally, for projects with very aggressive time schedules, RAD model should be preferred.
iv. Large software: Only for software supporting many features (large software) can incremental development and delivery be meaningfully carried out.

3.7.3. Application characteristics that render RAD unsuitable

i. Generic products (wide distribution)
ii. *Requirement of optimal performance and/or reliability*: For certain categories of products, optimal performance or reliability is required. Examples of such systems include an operating system (high reliability required) and a flight simulator software (high performance required). If such systems are to be developed using the RAD model, the desired product performance and reliability may not be realised.
iii. *Lack of similar products*: If a company has not developed similar software, then it would hardly be able to reuse much of the existing artifacts. In the absence of sufficient plug-in components, it becomes difficult to develop rapid prototypes through reuse, and use of RAD model becomes meaningless.
iv. *Monolithic entity*: For certain software, especially small-sized software, it may be hard to divide the required features into parts that can be incrementally developed and delivered. In this case, it becomes difficult to develop a software incrementally.

3.7.4. RAD versus iterative waterfall model

In the iterative waterfall model, all the functionalities of a software are developed together. On the other hand, in the RAD model the product functionalities are developed incrementally through heavy code and design reuse. Further, in the RAD model customer feedback is obtained on the developed prototype after each iteration and based on this the prototype is refined. Thus, it becomes easy to accommodate any request for requirements changes. However,

the iterative waterfall model does not support any mechanism to accommodate any requirement change requests. The iterative waterfall model does have some important advantages that include the following. Use of the iterative waterfall model leads to production of good quality documentation which can help during software maintenance. Also, the developed software usually has better quality and reliability than that developed using RAD.

3.7.5. RAD versus evolutionary model

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

3.8. AGILE DEVELOPMENT MODELS

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

Agile methods universally rely on an incremental approach to software specification, development, and delivery. They are best suited to application development where the system requirements usually change rapidly during the development process. They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system. They aim to cut down on process bureaucracy by avoiding work that has dubious long-term value and eliminating documentation that will probably never be used.

In the agile model, the requirements are decomposed into many small parts that can be incrementally developed. The agile model adopts an iterative approach. Each incremental part is developed over an iteration. Each iteration is intended to be small and easily manageable and lasting for a couple of weeks only. At a time, only one increment is planned, developed, and then deployed at the customer site. No long-term plans are made. The time to complete an iteration is called a time box. The implication of the term time box is that the end date for an iteration does not change.

3.8.1. Philosophy of Agile Models

The philosophy behind agile methods is reflected in the agile manifesto that was agreed on by many of the leading developers of these methods. This manifesto states:

> *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> *Individuals and interactions over processes and tools*
>
> *Working software over comprehensive documentation*
>
> *Customer collaboration over contract negotiation*
>
> *Responding to change over following a plan*
>
> *That is, while there is value in the items on the right, we value the items on the left more*

3.8.2. Reason for Agile Model Necessity

In the traditional iterative waterfall-based software development models, the requirements for the system are determined at the start of a development project and are assumed to be fixed from that point on. Later changes to the requirements after the SRS document has been completed are discouraged. If at all any later requirement changes becomes unavoidable, then the cost of accommodating it becomes prohibitively high. On the other hand, accumulated experience indicates that customers frequently change their requirements during the development period due to a variety of reasons.

Customised applications (services) has become common place and the sales revenue generated worldwide from services already exceeds that of the software products. Clearly, iterative waterfall model is not suitable for development of such software. Since customization essentially involves reusing most of the parts of an existing application and consists of only carrying out minor modifications by writing minimal amounts of code. For such development projects, the need for more appropriate development models was deeply felt, and many researchers started to investigate in this direction.

Waterfall model is called a heavy weight model, since there is too much emphasis on producing documentation and usage of tools. This is often a source of inefficiency and causes the project completion time to be much longer in comparison to the customer expectations.

Waterfall model prescribes almost no customer interactions after the requirements have been specified. In fact, in the waterfall model of software development, customer interactions are largely confined to the project initiation and project completion stages.

The agile software development model was proposed in the mid-1990s to overcome the serious shortcomings of the waterfall model of development identified above. The agile model was primarily designed to help a project to adapt to change requests quickly. 1Thus, a major aim of the agile models is to facilitate quick project completion. But, how is agility achieved in these models? Agility is achieved by fitting the process to the project, i.e. removing activities that may not be necessary for a specific project. Also, anything that that wastes time and effort is avoided.

3.8.3. Agile Models

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

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

3.8.4. Principle of Agile Development

i. *Customer involvement*: Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.

ii. *Incremental delivery*: The software is developed in increments with the customer specifying the requirements to be included in each increment.

iii. *People not process*: The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.

iv. *Embrace change*: Expect the system requirements to change and so design the system to accommodate these changes

v. *Maintain simplicity*: Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

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

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

3.8.5. Comparison between Agile and Other Models

Agile model versus iterative waterfall model

The waterfall model is highly structured and systematically steps through requirements-capture, analysis, specification, design, coding, and testing stages in a planned sequence. Progress is generally measured in terms of the number of completed and reviewed artifacts such as requirement specifications, design documents, test plans, code reviews, etc. for which review is complete. In contrast, while using an agile model, progress is measured in terms of the developed and delivered functionalities. In agile model, delivery of working versions of a software is made in several increments. However, as regards to similarity it can be said that agile teams use the waterfall model on a small scale, repeating the entire waterfall cycle in every iteration. If a project being developed using waterfall model is cancelled mid-way during development, then there i s nothing to show from the abandoned project beyond several documents. With agile model, even if a project is cancelled midway, it still leaves the customer with some worthwhile code that might possibly have already been put into live operation.

Agile versus exploratory programming

Though a few similarities do exist between the agile and exploratory programming styles, there are vast differences between the two as well. Agile development

model's frequent re- evaluation of plans, emphasis on face-to-face communication, and relatively sparse use of documentation are similar to that of the exploratory style. Agile teams, however, do follow defined and disciplined processes and carry out systematic requirements capture, rigorous designs, compared to chaotic coding in exploratory programming.

Agile model versus RAD model

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

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

CHAPTER FOUR

SOFTWARE PROJECT MANAGEMENT

Software project management is an essential part of software engineering. Projects need to be managed because professional software engineering is always subject to organizational budget and schedule constraints. The project manager's job is to ensure that the software project meets and overcomes these constraints as well as delivering high-quality software. Good management cannot guarantee project success. However, bad management usually results in project failure: the software may be delivered late, cost more than originally estimated, or fail to meet the expectations of customers.

The success criteria for project management obviously vary from project to project but, for most projects, important goals are:

1. Deliver the software to the customer at the agreed time.
2. Keep overall costs within budget.
3. Deliver software that meets the customer's expectations.
4. Maintain a happy and well-functioning development team.

4.2. Difference between software Development Engineering and other Engineering

Software engineering is different from other types of engineering in a number of ways that make software management particularly challenging. Some of these differences are:

1. *The product is intangible*: A manager of a shipbuilding or a civil engineering project can see the product being developed. If a schedule slips, the effect on the product is visible—parts of the structure are obviously unfinished. Software is intangible. It cannot be seen or touched. Software project managers cannot see progress by simply looking at the artifact that is being constructed. Rather, they rely on others to produce evidence that they can use to review the progress of the work.
2. *Large software projects are often 'one-off' projects*: Large software projects are usually different in some ways from previous projects. Therefore, even managers who have a large body of previous experience may find it difficult to anticipate problems. Furthermore, rapid technological changes in computers and communications can make a manager's experience

obsolete. Lessons learned from previous projects may not be transferable to new projects.

3. *Software processes are variable and organization-specific*: The engineering process for some types of system, such as bridges and buildings, is well understood. However, software processes vary quite significantly from one organization to another. Although there has been significant progress in process standardization and improvement, we still cannot reliably predict when a particular software process is likely to lead to development problems. This is especially true when the software project is part of a wider systems engineering project.

4. *Exactness of the solution*: Mechanical components such as nuts and bolts typically work satisfactorily as long as they are within a tolerance of 1 per cent or so of their specified sizes. However, the parameters of a function call in a program are required to be in complete conformity with the function definition. This requirement not only makes it difficult to get a software product up and working, but also makes reusing parts of one software product in another difficult. This requirement of exact conformity of the parameters of a function introduces additional risks and contributes to the complexity of managing software projects.

5. *Team-oriented and intellect-intensive work*: Software development projects are akin to research projects in the sense that they both involve team-oriented, intellect-intensive work. In contrast, projects in many domains are labour-intensive and each member works in a high degree of autonomy. Examples of such projects are planting rice, laying roads, assembly-line manufacturing, constructing a multi-storeyed building, etc. In a software development project, the life cycle activities not only highly intellectintensive, but each member has to typically interact, review, and interface with several other members, constituting another dimension of complexity of software projects.

4.3. Roles of Software Manager

It is impossible to write a standard job description for a software project manager. The job varies tremendously depending on the organization and the software product being developed. However, most managers take responsibility at some stage for some or all of the following activities:

1. *Project planning* Project managers are responsible for planning, estimating and scheduling project development, and assigning people to tasks.

They supervise the work to ensure that it is carried out to the required standards and monitor progress to check that the development is on time and within budget.

2. *Reporting* Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software. They have to be able to communicate at a range of levels, from detailed technical information to management summaries. They have to write concise, coherent documents that abstract critical information from detailed project reports. They must be able to present this information during progress reviews.

3. *Risk management* Project managers have to assess the risks that may affect a project, monitor these risks, and take action when problems arise. 4. People management Project managers are responsible for managing a team of people. They have to choose people for their team and establish ways of working that lead to effective team performance.

5. *Proposal writing:* The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out. It usually includes cost and schedule estimates and justifies why the project contract should be awarded to a particular organization or team. Proposal writing is a critical task as the survival of many software companies depends on having enough proposals accepted and contracts awarded. There can be no set guidelines for this task; proposal writing is a skill that you acquire through practice and experience.

4.4. Organisation of the software project management plan (SPMP) document

1. Introduction
    (a) Objectives
    (b) Major Functions
    (c) Performance Issues
    (d) Management and Technical Constraints
2. Project estimates
    (a) Historical Data Used
    (b) Estimation Techniques Used
    (c) Effort, Resource, Cost, and Project Duration Estimates
3. Schedule
    (a) Work Breakdown Structure
    (b) Task Network Representation

(c) Gantt Chart Representation
(d) PERT Chart Representation

4. Project resources
   (a) People
   (b) Hardware and Software
   (c) Special Resources
5. Staff organisation
   (a) Team Structure
   (b) Management Reporting
6. Risk management plan
   (a) Risk Analysis
   (b) Risk Identification
   (c) Risk Estimation
   (d) Risk Abatement Procedures
7. Project tracking and control plan
   (a) Metrics to be tracked
   (b) Tracking plan
   (c) Control plan
8. Miscellaneous plans
   (a) Process Tailoring
   (b) Quality Assurance Plan
   (c) Configuration Management Plan
   (d) Validation and Verification
   (e) System Testing Plan
   (f) Delivery, Installation, and Maintenance Plan

4.4. RISK MANAGEMENT

Risk management is one of the most important jobs for a project manager. Risk management involves anticipating risks that might affect the project schedule or the quality of the software being developed, and then taking action to avoid these risks (Hall, 1998; Ould, 1999).

Risk management aims at reducing the chances of a risk becoming real as well as reducing the impact of a risks that becomes real. Risk management consists of three essential activities—risk identification, risk assessment, and risk mitigation. We discuss these three activities in the following subsection. This essential activities are represented in Figure 4.1 with their brief description.

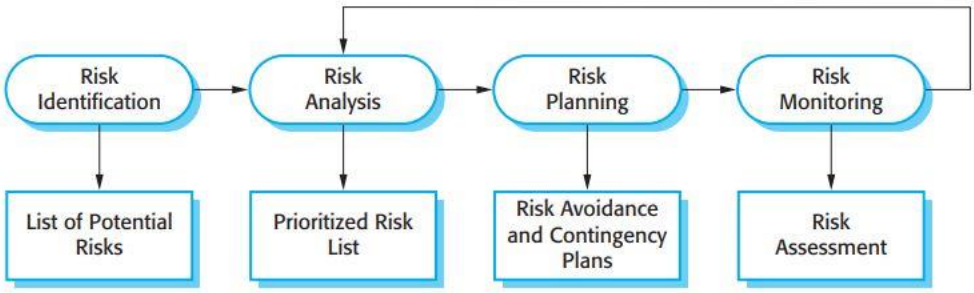

Figure 4.1: Risk Management Processes

Risks may threaten the project, the software that is being developed, or the organization. There are, therefore, three related categories of risk:

1. *Project risks*: Risks that affect the project schedule or resources. An example of a project risk is the loss of an experienced designer. Finding a replacement designer with appropriate skills and experience may take a long time and, consequently, the software design will take longer to complete.
2. *Product risks*: Risks that affect the quality or performance of the software being developed. An example of a product risk is the failure of a purchased component to perform as expected. This may affect the overall performance of the system so that it is slower than expected.
3. *Business risks*: Risks that affect the organization developing or procuring the software. For example, a competitor introducing a new product is a business risk. The introduction of a competitive product may mean that the assumptions made about sales of existing software products may be unduly optimistic.

Table 4.1: Common Risk that affect various aspect of software development

RISK	**ASPECT**	**DESCRIPTION**
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organizational management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.

Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated
Technology change	Business	The underlying technology on which the system is built is superseded by new technology
Product competition	Business	A competitive product is marketed before the system is completed.

4.4.1. Risk Identification

Risk identification is the first stage of the risk management process. It is concerned with identifying the risks that could pose a major threat to the software engineering process, the software being developed, or the development organization. Risk identification may be a team process where a team get together to brainstorm possible risks. Alternatively, the project manager may simply use his or her experience to identify the most probable or critical risks. As a starting point for risk identification, a checklist of different types of risk may be used. There are at least six types of risk that may be included in a risk checklist:

1. Technology risks: Risks that derive from the software or hardware technologies that are used to develop the system.

2. People risks: Risks that are associated with the people in the development team.

3. Organizational risks: Risks that derive from the organizational environment where the software is being developed.

4. Tools risks: Risks that derive from the software tools and other support software used to develop the system.

5. Requirements risks: Risks that derive from changes to the customer requirements and the process of managing the requirements change.

6. Estimation risks: Risks that derive from the management estimates of the resources required to build the system

4.4.2. Risk Analysis

During the risk analysis process, you have to consider each identified risk and make a judgment about the probability and seriousness of that risk. There is no easy way to do this. You have to rely on your own judgment and experience of previous projects and the problems that arose in them. The different types of risks identified are categorized based on the possible examples of risks faced in software development that might fall under any of the aforementioned categories. This is explicitly presented in Table 4.2

It is not possible to make precise, numeric assessment of the probability and seriousness of each risk. Rather, you should assign the risk to one of a number of bands:

1. The probability of the risk might be assessed as very low (10%), low (10–25%), moderate (25–50%), high (50–75%), or very high (75%).

2. The effects of the risk might be assessed as catastrophic (threaten the survival of the project), serious (would cause major delays), tolerable (delays are within allowed contingency), or insignificant.

Table 4.2: Risks and their possible examples

Types of Risk	Examples of possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. (1) Reusable software components contain defects that mean they cannot be reused as planned. (2)
People	It is impossible to recruit staff with the skills required. (3) Key staff are ill and unavailable at critical times. (4) Required training for staff is not available. (5)
Organizational	The organization is restructured so that different management are responsible for the project. (6) Organizational financial problems force reductions in the project budget. (7)
Tools	The code generated by software code generation tools is inefficient. (8) Software tools cannot work together in an integrated way. (9)
Requirements	Changes to requirements that require major design rework are proposed. (10) Customers fail to understand the impact of requirements changes. (11)
Estimation	The time required to develop the software is underestimated. (12)

The rate of defect repair is underestimated.
(13) The size of the software is underestimated. (14

4.4.3. Risk Planning

The risk planning process considers each of the key risks that have been identified, and develops strategies to manage these risks. For each of the risks, you have to think of actions that you might take to minimize the disruption to the project if the problem identified in the risk occurs. You also should think about information that you might need to collect while monitoring the project so that problems can be anticipated.

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of a program generator.

Figure 4.2: Different risk strategies.

These strategies fall into three categories:

1. Avoidance strategies: Following these strategies means that the probability that the risk will arise will be reduced. An example of a risk avoidance strategy is the strategy for dealing with defective components shown in Figure 4.2.

2. Minimization strategies: Following these strategies means that the impact of the risk will be reduced. An example of a risk minimization strategy is the strategy for staff illness shown in Figure 4.2

3. Contingency plans: Following these strategies means that you are prepared for the worst and have a strategy in place to deal with it. An example of a contingency strategy is the strategy for organizational financial problems that I have shown in Figure 4.2.

4.4.4. Risk Monitoring

Risk monitoring is the process of checking that your assumptions about the product, process, and business risks have not changed. You should regularly assess each of the identified risks to decide whether or not that risk is becoming more or less probable. You should also think about whether or not the effects of the risk have changed. To do this, you have to look at other factors, such as the number of requirements change requests, which give you clues about the risk probability and its effects. These factors are obviously dependent on the types of risk. Figure 22.6 gives some examples of factors that may be helpful in assessing these risk types. You should monitor risks regularly at all stages in a project. At every management review, you should consider and discuss each of the key risks separately. You should decide if the risk is more or less likely to arise and if the seriousness and consequences of the risk have changed.

PART TWO

COMPUTATIONAL TECHNIQUES

CHAPTER FIVE

ALGORITHM DEVELOPMENT

References

Budgen, D. (2003). Software Design (2nd Edition). Harlow, UK.: Addison-Wesley.

Hall, E. (1998). Managing Risk: Methods for Software Systems Development. Reading, Mass.: Addison-Wesley.

Mall, R. (2014). Fundamentals of Software Engineering *4th edition*. PHI Learning, Private Limited Delhi

Ould, M. (1999). Managing Software Quality and Business Risk. Chichester: John Wiley & Sons.

Royce, W. W. (1970). 'Managing the Development of Large Software Systems: Concepts and Techniques'. IEEE WESTCON, Los Angeles CA: 1–9

Sommerville, I. (2011) Software Engineering *9th edition*. Pearson Education, Inc., publishing as Addison-Wesley