SOFTWARE ENGINEERING

By

Dr. Jayanta Mondal

KIIT Deemed to be University

Syllabus

Software Process Models	Software product, Software crisis, Handling complexity through Abstraction and Decomposition, Overview of software development activities. **Process Models: Classical waterfall model, iterative waterfall model**, prototyping model, evolutionary model, spiral model, RAD model. Agile models: Extreme programming and Scrum.
Software Requirement Engineering	Requirement Gathering and analysis, Functional and non functional requirements, Software Requirement Specification(SRS) , IEEE 830 guidelines, Decision tables and trees.
Software Project Management	Responsibilities of a Software project manager, project planning, Metrics for project size estimation, Project estimation techniques, Empirical estimation techniques, COCOMO models, Scheduling, Organization & team structure, Staffing,.
Structural Analysis & Design	Overview of design process : High level and detailed design, Cohesion & coupling, Modularity and layering, Function–Oriented software design: Structural Analysis, Structural Design (DFD and Structured Chart), Object Oriented Analysis & Design, Command language, menu and iconic interfaces.
Testing Strategies	Coding, Code Review, Documentation, Testing: - Unit testing, Black-box Testing, White-box testing, Cyclomatic complexity measure, Coverage analysis, Debugging, Integration testing, System testing, Regression testing.
Software Reliability Software Maintenance	Software reliability, reliability measures, reliability growth modelling, Quality SEI CMM, Characteristics of software maintenance, software reverse engineering, software re engineering, Software reuse
Emerging Topics	Client-Server Software engineering, Service Oriented Architecture (SOA), Software as a Service (SaaS)

The Disadvantage of *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 small 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.

The right way ahead:

- A long term goal
- A proper planning
- Short term goals

How to achieve the Goal?

- By adhering a right process.
- A software Process: set of related activities that leads to the production of a software.
- Software Engineering is a systematic approach to design, develop, implement and maintain a software.

A FEW BASIC CONCEPTS

- **Software life cycle:** Based on the 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.
- **Inception stage**: 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.

A FEW BASIC CONCEPTS

- **Maintenance Phase:** 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.

A FEW BASIC CONCEPTS

- **Retirement Phase:** 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 (initial phase to retirement phase) through which it evolves during its life time.

Software Development Lifecycle 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 describes the different activities that need to be carried out for the software to evolve in its life cycle."

Why document a development process?

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

Phase entry and exit criteria

- A good SDLC besides clearly identifying the different phases in the life cycle, should unambiguously define the entry and exit criteria for each phase. The phase entry (or exit) criteria is usually expressed as a set of conditions that needs to be satisfied for the phase to start (or to complete).

As an example, the phase exit criteria for the software requirements specification phase, can be that the *software requirements specification* (SRS) document is ready, has been reviewed internally, and also has been reviewed and approved by the customer. Only after these criteria are satisfied, the next phase can start.

Different Activities in a SDLC Model

- Feasibility Study
- Requirement Analysis and Specification
- Design
- Coding & Unit Testing
- Integration & System Testing
- Implementation
- Maintenance

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,

  and various other constraints on the development.

Importance of **Feasibility study**

The collected data during Feasibility Study are analyzed to perform the following:

- **Development of an overall understanding of the problem:** It is necessary to first develop an overall understanding of what the customer requires to be developed. For this, only the important requirements of the customer need to be understood and the details of various requirements
- **Formulation of the various possible strategies for solving the problem:** various possible high-level solution schemes to the problem are determined.
- **Evaluation of the different solution strategies:** The different identified solution schemes are analyzed to evaluate their benefits and shortcomings.

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.
- **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.
- **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.

Software Requirements Specification (SRS) Document

- The SRS document is written using end-user terminology. This makes the SRS document understandable to the customer. Understandability of the SRS document is an important issue.
- The SRS document normally serves as a contract between the development team and 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.
- **Procedural 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.
- **Object-oriented design approach:** 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 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.

Integration and system testing

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

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

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

β-testing: This is the system testing performed by a friendly set of customers.

Acceptance testing: After the software has been delivered, the customer performs system testing to determine whether t o accept the delivered software or to reject it.

Implementation/Installation

- Check if the hardware at the clients side, if required then upgrade.
- Software is installed in the client machine.
- If required, proper training is given to the users.

Maintenance

- The total effort spent on maintenance of a typical software during its operation phase is much more than that required for developing the software itself.

Corrective maintenance: This type of maintenance is carried out to correct errors that were not discovered during the product development phase.

Perfective maintenance: This type of maintenance is carried out to improve the performance of the system, or to enhance the functionalities of the system based on customer's requests.

Adaptive maintenance: Adaptive maintenance is usually required for porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system.

WATERFALL MODEL

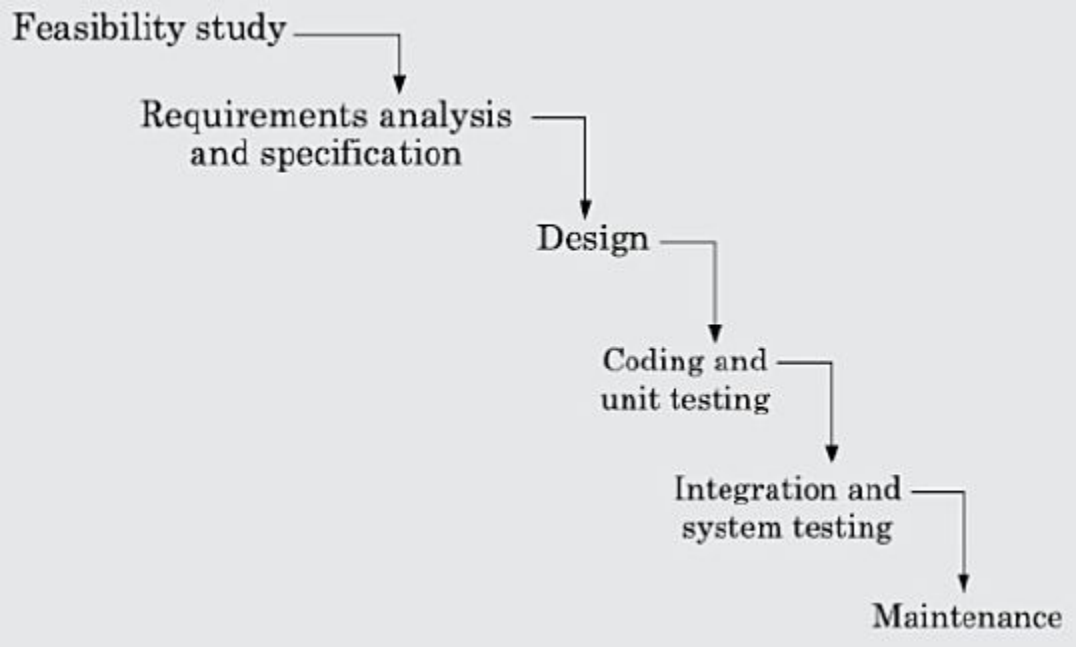
- The waterfall model and its derivatives were extremely popular in the 1970s and still are heavily being used across many development projects.
- The waterfall model is possibly the most obvious and intuitive way in which software can be developed through team effort.
- We can think of the waterfall model as a generic model that has been extended in many ways.

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.

Why it is called The classical waterfall model?

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



- **Figure 2.1:** Classical waterfall model.

Phases of the classical waterfall model

- The different phases 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.*
- In the waterfall model, different life cycle phases typically require relatively different amounts of efforts to be put in by the development team.

Relative effort distribution among different phases

- The relative amounts of effort spent on different phases for a typical software has been shown in Figure 2.2.

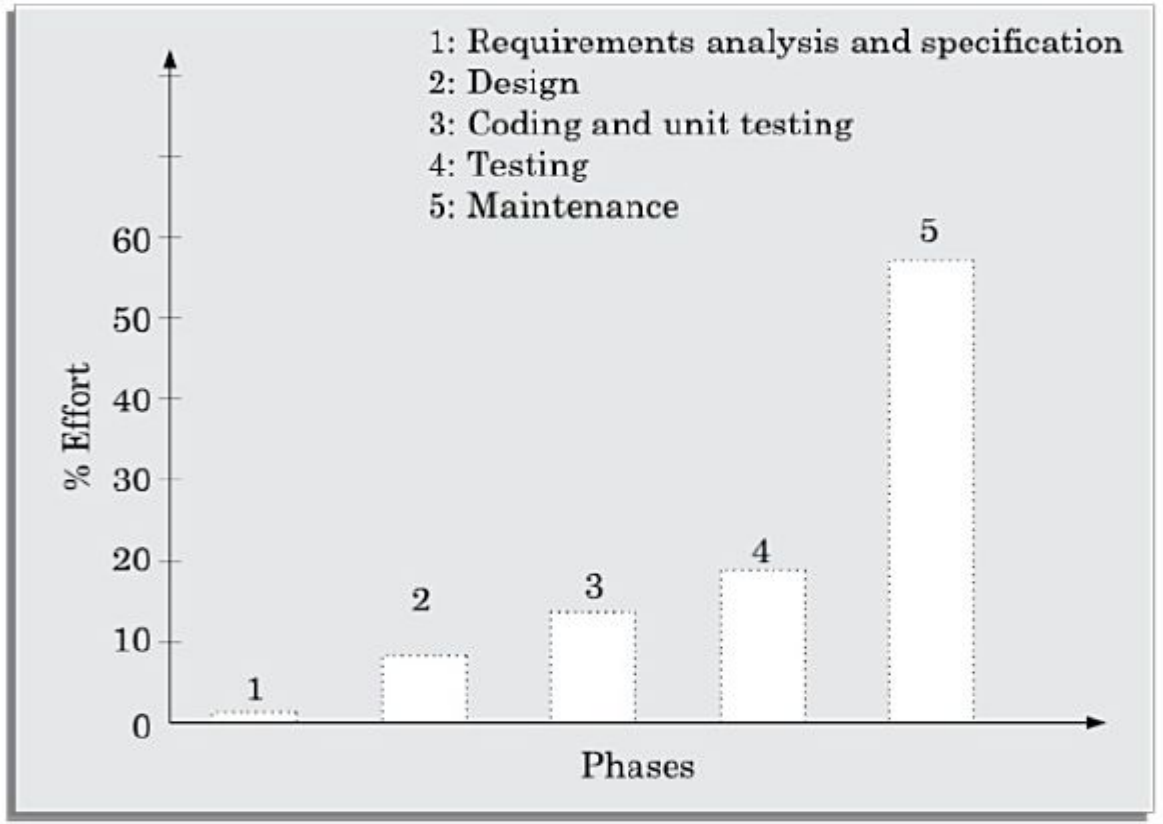


Figure 2.2: Relative effort distribution among different phases of a typical product.

Shortcomings of the classical waterfall model

- **No feedback paths:** The activities carried out in it and any artifacts produced in this phase are considered to be final and are closed for any rework and incorporates no mechanism for error correction.

- **Difficult to accommodate change requests:** 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.

Shortcomings of the classical waterfall model

- **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.

Is the classical waterfall model useful at all?

- It is hard to use the classical waterfall model in real projects. In any practical development environment, as the software takes shape, several iterations through the different waterfall stages become necessary for correction of errors committed during various phases.

Therefore, the classical waterfall model is hardly usable for software development.

But, as suggested by Parnas [1972] the final documents for the product should be written as if the product was developed using a pure classical waterfall.

Iterative Waterfall Model

- 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 allow for correcting errors committed by a programmer during some phase, as and when these are detected in a later phase.

Diagram of Iterative Waterfall Model

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.

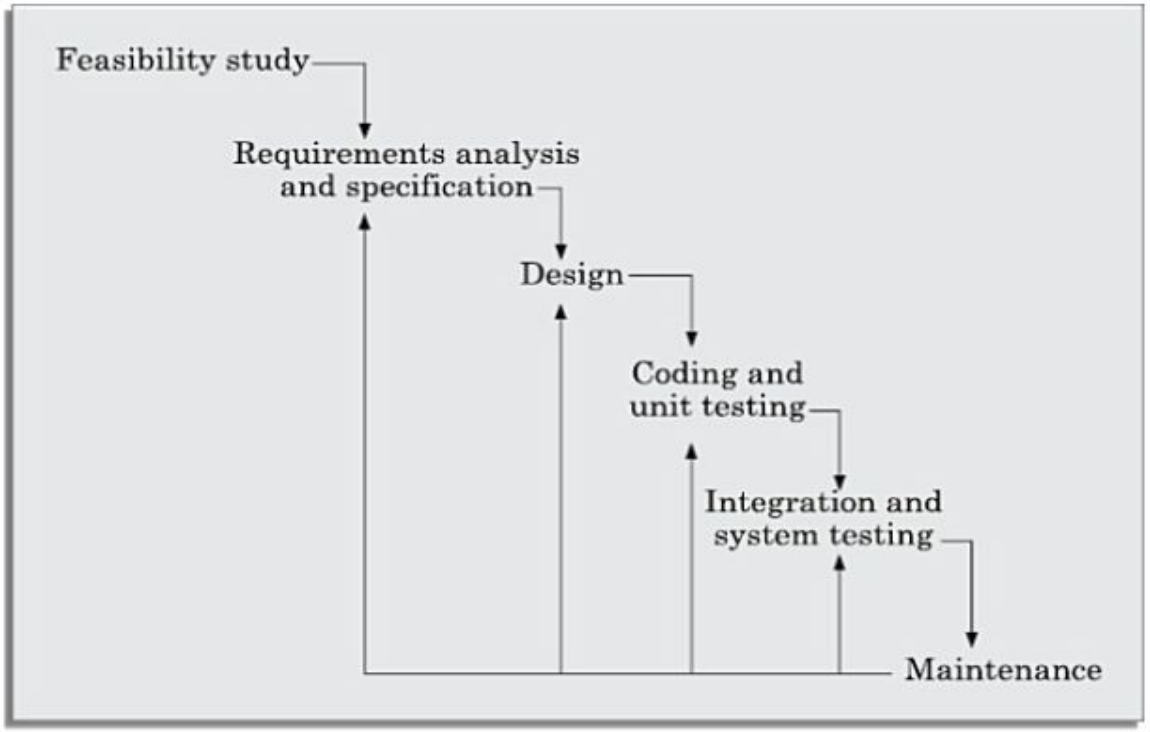


Figure 2.3: Iterative waterfall model.

Shortcomings of the iterative waterfall model

- **Difficult to accommodate change requests:** Once requirements have been frozen, the waterfall model provides no scope for any modifications to the requirements.
- **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.

Shortcomings of the iterative waterfall model

- **Phase overlap not supported:** Strict adherence to the waterfall model creates *blocking states*.
- **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.
- **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.

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.
- The principle of detecting errors as close to their points of commitment as possible is known as *phase containment of errors*.
- A popular technique is to rigorously review the documents produced at the end of a phase.

Phase overlap

- In practice the activities of different phases overlap (as shown in Figure 2.4) is necessary due to two main reasons:

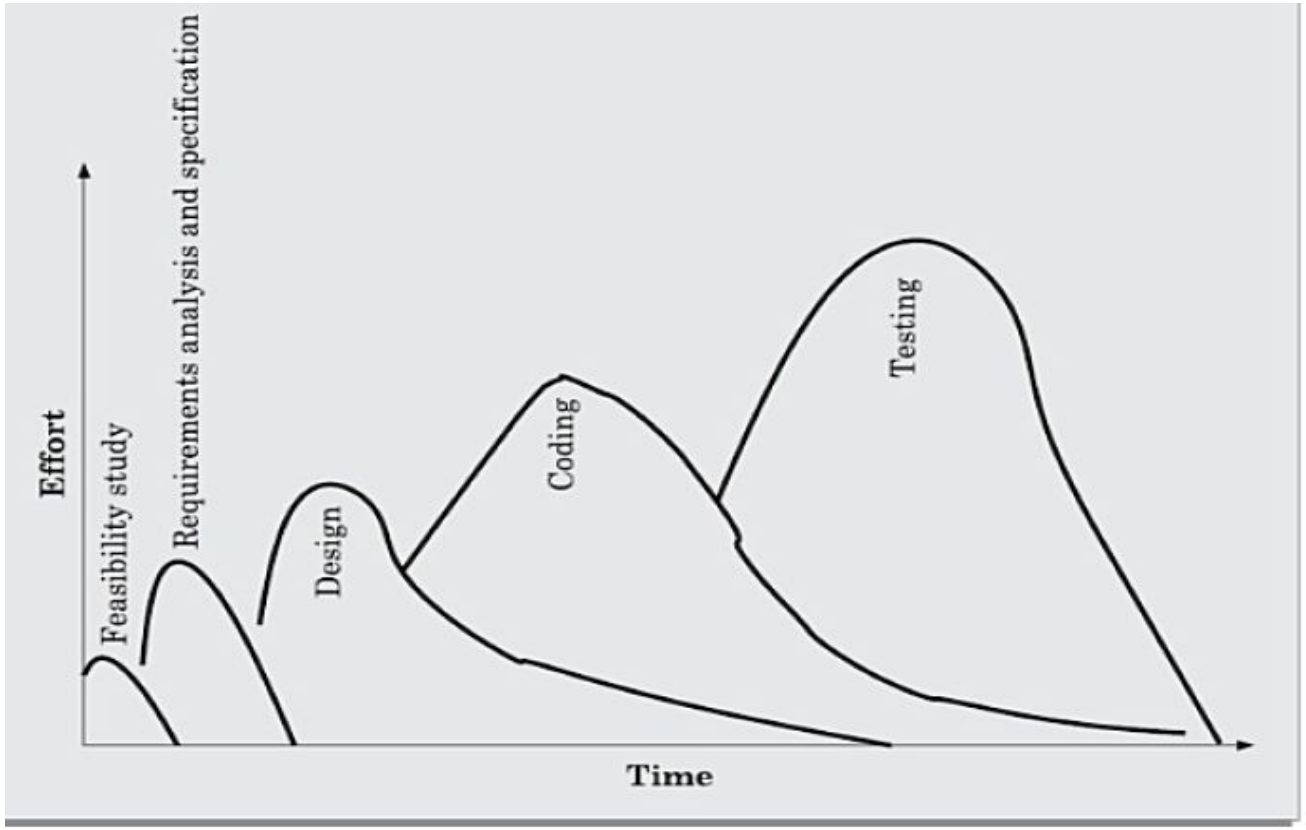


Figure 2.4: Distribution of effort for various phases in the iterative waterfall model.

Phase overlap

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

Thank you

If you have any doubts …. Feel free to ask