Software Engineering (CSC 420)

Module II

Introduction

The role of software engineering cannot be neglected in the field of software development. The advent of computers introduced the need for software and the quality of software introduced the need for software engineering. Software engineering has come a long way since 1968, when the term was first used at a NATO conference, and software itself has entered our lives in ways that few had anticipated, even a decade ago. So a firm grounding in software-engineering theory and practice is essential for understanding how to build good error-free software at an inexpensive price and with less time and for evaluating the risks and opportunities that software presents in our everyday lives.

1. Software and Software Engineering

The term ***software engineering*** is composed of two words, Software and Engineering.

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

software engineering can be defined as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product. Other popular definitions of software engineering are:

• Software engineering is concerned with the theories, methods and tools for developing, managing and evolving software products.

– I. Sommerville

• The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them

– B.W.Boehm

Why the use of software engineering in software development?

The use of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

- **Large software** - It is easier to build a wall than to build a house or building, likewise, as the size of software become large engineering has to step in to give it a scientific process.
- **Scalability-** If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.
- **Cost-** As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.
- **Dynamic Nature-** The always growing and adapting nature of software hugely depends upon the environment in which the user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.
- **Quality Management-** Better process of software development provides better and quality software product.

CHARACTERESTICS OF GOOD SOFTWARE

A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

1. Operational
2. Transitional
3. Maintenance

Well-engineered and crafted software is expected to have the following characteristics:

Operational

This tells us how well software works in operations. It can be measured on:

- Budget
- Usability
- Efficiency
- Correctness
- Functionality
- Dependability
- Security
- Safety

Transitional

This aspect is important when the software is moved from one platform to another:

- Portability
- Interoperability
- Reusability
- Adaptability

Maintenance

This aspect briefs about how well a software has the capabilities to maintain itself in the ever-changing environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products

2. Software development life-cycle

The software-development life-cycle is used to facilitate the development of a large software product in a systematic, well-defined, and cost-effective way. An information system goes through a series of phases from conception to implementation. This process is called the Software-Development Life-Cycle as illustrated in figure 1a.

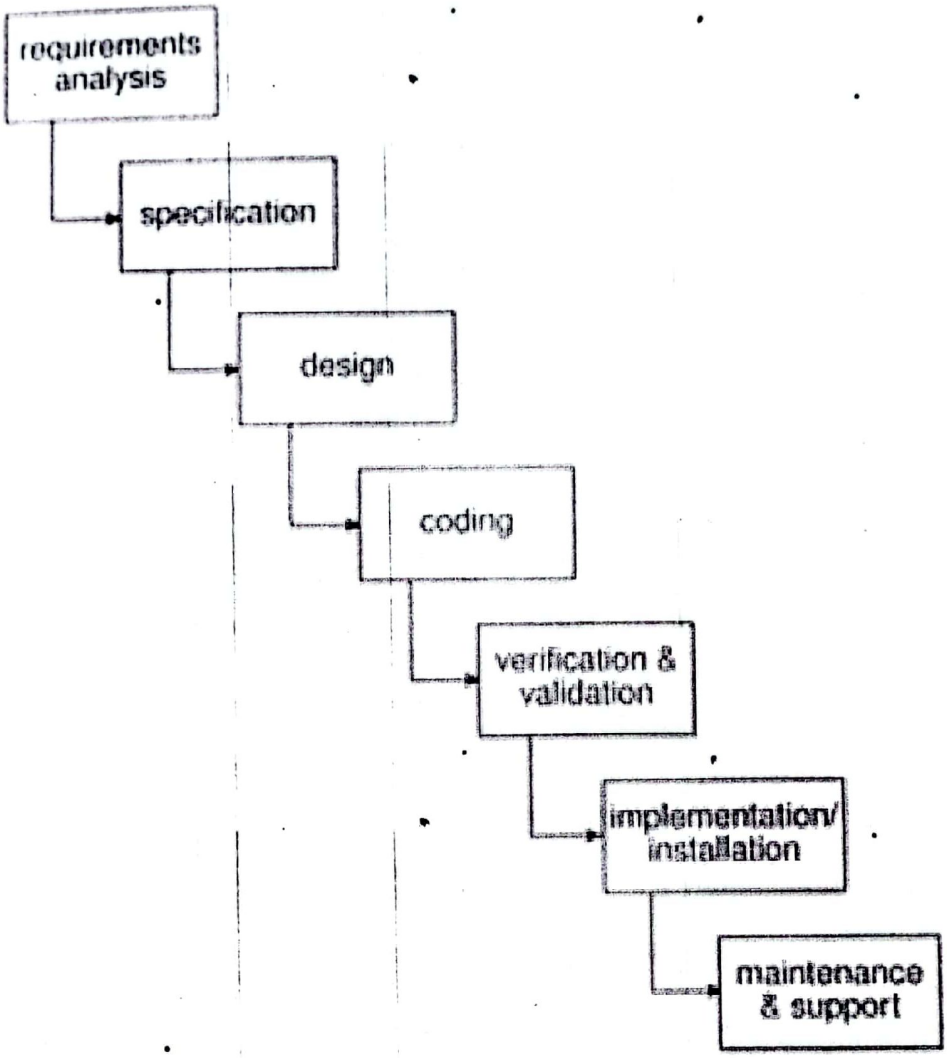


Figure 1a: A typical project life-cycle

Various reasons for using a life-cycle model include:

- Helps to understand the entire process
- Enforces a structured approach to development
- Enables planning of resources in advance
- Enables subsequent controls of these resources
- Aids management to track progress of the system

The software development life-cycle consists of several phases and these phases need to be identified along with defining the entry and exit criteria for every phase. A phase can begin only when the corresponding phase-entry criteria are satisfied. Similarly, a phase can be considered to be complete only when the corresponding exit criteria are satisfied. If there is no clear indication of the entry and exit for every phase, it becomes very difficult to track the progress of the project.

The software development life-cycle can be divided into 5-9 phases, i.e., it must have a minimum of five phases and a maximum of nine phases. On average it has seven or eight phases. These are:

- Project initiation and planning/Recognition of need/Preliminary investigation
- Project identification and selection/Feasibility study
- Project analysis
- System design
- Coding
- Testing
- Implementation
- Maintenance

1. Recognition of Need. Recognition of need is nothing but the problem definition. It is the decision about problems in the existing system and the impetus for system change. The first stage of any project or system-development life-cycle is called the preliminary investigation. It is a brief investigation of the system under consideration. This investigation provides the organization's computer steering committee and any project team a set of terms or references for more detailed work. This is carried out by a senior manager and will result in a study proposal. At this stage the need for changes in the existing system are identified and shortcomings of the existing system are detected. These are stated clearly providing the basis for the initial or feasibility study.

2. **Feasibility Study.** A feasibility study is a preliminary study which investigates the information needs of prospective users and determines the resource requirements, costs, benefits, and feasibility of a proposed project. The goal of feasibility studies is to evaluate

alternative systems and to propose the most feasible and desirable systems for development. The feasibility of a proposed system can be evaluated in terms of four major categories:

(*i*) *Organizational Feasibility.* Organizational feasibility is how well a proposed information system supports the objectives of the organization and is a strategic plan for an information system. For example, projects that do not directly contribute to meeting an organization's strategic objectives are typically not funded.

(*ii*) *Economic Feasibility.* Economic feasibility is concerned with whether expected cost savings, increased revenue, increased profits, reductions in required investments, and other types of benefits will exceed the costs of developing and operating a proposed system. For example, if a project can't cover its development costs, it won't be approved, unless mandated by government regulations or other considerations.

(*iii*) *Technical Feasibility.* Technical feasibility can be demonstrated if reliable hardware and software capable of meeting the needs of a proposed system can be acquired or developed by the business in the required time.

(*iv*) *Operational Feasibility.* Operational feasibility is the willingness and ability of management, employees, customers, suppliers, and others to operate, use, and support a proposed system. For example, if the software for a new system is too difficult to use, employees may make too many errors and avoid using it. Thus, it would fail to show operational feasibility.

3. **Project Analysis.** Project analysis is a detailed study of the various operations performed by a system and their relationships within and outside the system. Detailed investigation should be conducted with personnel closely involved with the area under investigation, according to the precise terms of reference arising out of the initial study reports. The tasks to be carried out should be clearly defined such as:

- Examine and document the relevant aspects of the existing system, its shortcomings and problems.
- Analyze the findings and record the results.
- Define and document in an outline the proposed system.
- Test the proposed design against the known facts.
- Produce a detailed report to support the proposals.
- Estimate the resources required to design and implement the system.

The objectives at this stage are to provide solutions to stated problems, usually in the form of specifications to meet the user requirements and to make recommendations for a new computer-based system. Analysis is an iterative and progressive process, examining information flows and evaluating various alternative design solutions until a preferred solution is available. This is documented as the system proposal.

4. **System Design.** System design is the most creative and challenging phase of the system-development life-cycle. The term design describes the final system and process by which it is developed. Different stages of the design phase are shown in Figure 1b. This phase is a very important phase of the life-cycle. This is a creative as well as a technical activity including the following tasks:

- Appraising the terms of reference
- Appraising the analysis of the existing system, particularly problem areas
- Defining precisely the required system output
- Determining data required to produce the output
- Deciding the medium and opening the files
- Devising processing methods and using software to handle files and to produce output
- Determining methods of data capture and data input
- Designing the output forms

- Defining detailed critical procedures
- Calculating timings of processing and data movements
- Documenting all aspects of design

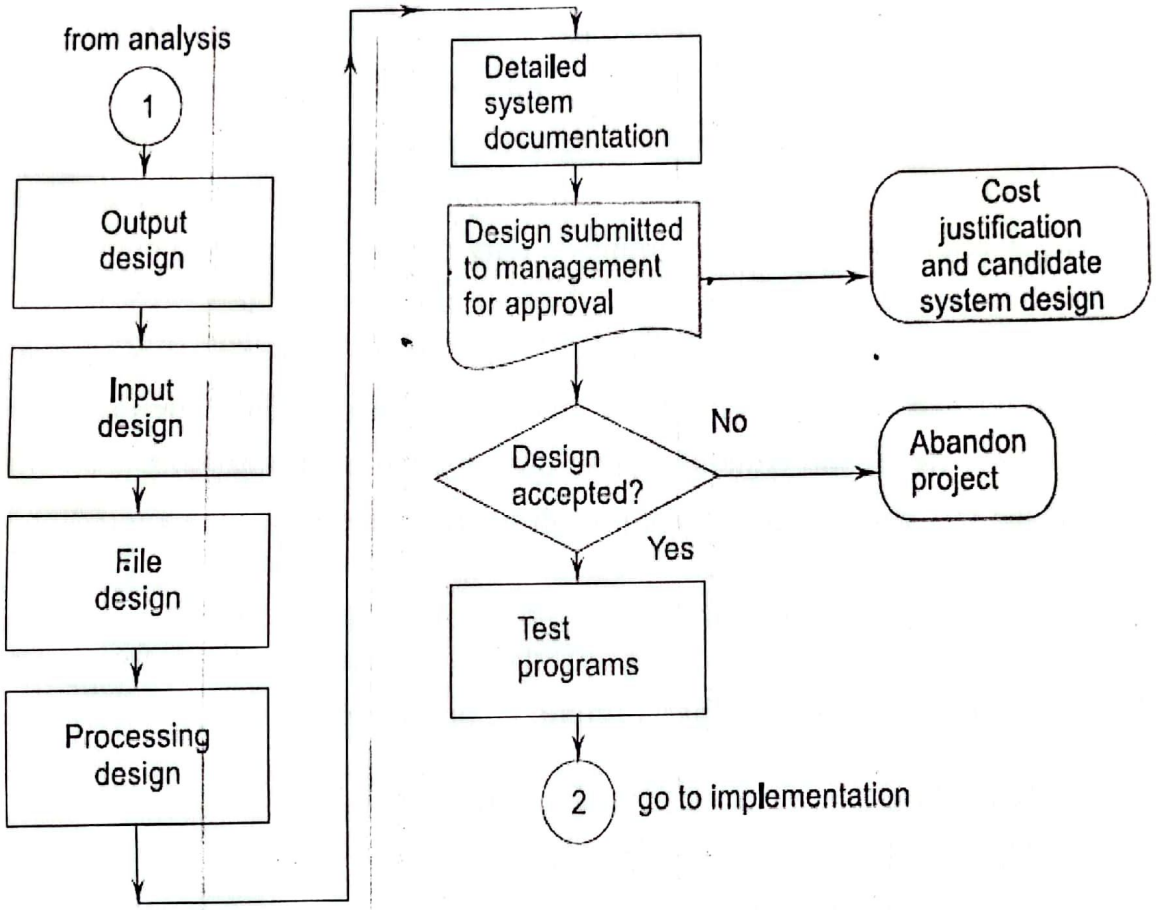


Fig. 1b: Cycle of Design Phase

5. **Coding.** The goal of the coding phase is to translate the design of the system into code in a given programming language. In this phase the aim is to implement the design in the best possible manner. This phase affects both testing and maintenance phases. Well-written code can reduce the testing and maintenance effort. Hence, during coding the focus is on developing programs that are easy to read and understand and not simply on developing programs that are simple to write. Coding can be subject to company-wide standards that may define the entire layout of programs, such as headers for comments in every unit, naming conventions for variables, classes and functions, the maximum number of lines in each component, and other aspects of standardization. Structured

programming helps to understand the flow of a program. The goal of structured programming is to linearize the control flow in the program. Single entry-single exit constructs should be used. The constructs include selection (if-then-else) and iteration (while, repeat-unit).

6. **Testing.** Testing is the major quality-control measure used during software development. Its basic function is to detect errors in the software. Thus, the goal of testing is to uncover requirement, design, and coding errors in the program. Testing is an extremely critical and time-consuming activity. It requires proper planning of the overall testing process. During the testing of the unit, the specified test cases are executed and the actual results are compared with the expected output. The final output of the testing phase is the test report and the error report, or a set of such reports (one for each unit tested). Each test report contains the set of test cases and the result of executing the code with these test cases. The error report describes the errors encountered and the action taken to remove the errors.

Testing cannot show the absence of defects; it can show only the software errors present. During the testing phase emphasis should be on the following:

- Tests should be planned long before testing begins.
- All tests should be traceable to customer requirements.
- Tracing should begin "in the small" and progress toward testing "in the large."
- For most effective testing, independent, third parties should conduct testing.

7. **Implementation.** The implementation phase is less creative than system design. It is mainly concerned with user training, site selection, and preparation and file conversion. Once the system has been designed, it is ready for implementation. Implementation is concerned with those tasks leading immediately to a fully operational system. It involve programmers, users, and operations management, but its planning and timing is a prime function of a systems analyst. It includes the final testing of the complete system to user satisfaction, and supervision of initial operation of the system. Implementation of the system also includes providing security to the system.

Types of Implementation

There are three types of implementation:

i. Implementation of a computer system to replace a manual system.
ii. Implementation of a new computer system to replace an existing one.
iii. Implementation of a modified application (software) to replace an existing one using the same computer.

8. **Maintenance**. Maintenance is an important part of the SDLC. If there is any error to correct or change then it is done in the maintenance phase. Maintenance of software is also a very necessary aspect related to software development. Many times maintenance may consume more time than the time consumed in the development. Also, the cost of maintenance varies from 50% to 80% of the total development cost.

Maintenance is not as rewarding or exciting as developing the systems. It may have problems such as:

- Availability of only a few maintenance tools.
- User may not accept the cost of maintenance.
- Standards and guidelines of project may be poorly defined.
- A good test plan is lacking.
- Maintenance is viewed as a necessary evil often delegated to junior programmers.
- Most programmers view maintenance as low-level drudgery.

3. Software Development Process Models

For the software development process, the goal is to produce a high-quality software product. It therefore focuses on activities directly related to production of the software, for example, design, coding, and testing. As the development process specifies the major development and quality control activities that need to be performed in the project, it

forms the core of the software process. The management process is often decided based on the development process. The Software development Process Models are:

i. Classical Waterfall Model
ii. Iterative Waterfall Model
iii. Prototyping Model
iv. Evolutionary Model
v. Spiral Model

A project's development process defines the tasks the project should perform, and the order in which they should be done. A process limits the degrees of freedom for a project by specifying what types of activities must be undertaken and in what order, such that the "shortest" (or the most efficient) path is obtained from the user needs to the software satisfying these needs. The process drives a project and heavily influences the outcome.

As discussed earlier, a process model specifies a general process, usually as a set of stages in which a project should be divided, the order in which the stages should be executed, and any other constraints and conditions on the execution of stages. The basic premise behind a process model is that, in the situations for which the model is applicable, using the process model as the project's process will lead to low cost, high quality, reduced cycle time, or provide other benefits. In other words, the process model provides generic guidelines for developing a suitable process for a project.

Due to the importance of the development process, various models have been proposed, some of the major models are:

1. Classical Waterfall Model

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it cannot be used in actual software development projects. Thus, this model can be considered to be a *theoretical way of developing software*. But all other development process models are essentially derived from the

classical waterfall model. So, in order to be able to appreciate other life cycle development process models, it is necessary to learn the classical waterfall model. Classical waterfall model divides the life cycle into the following phases

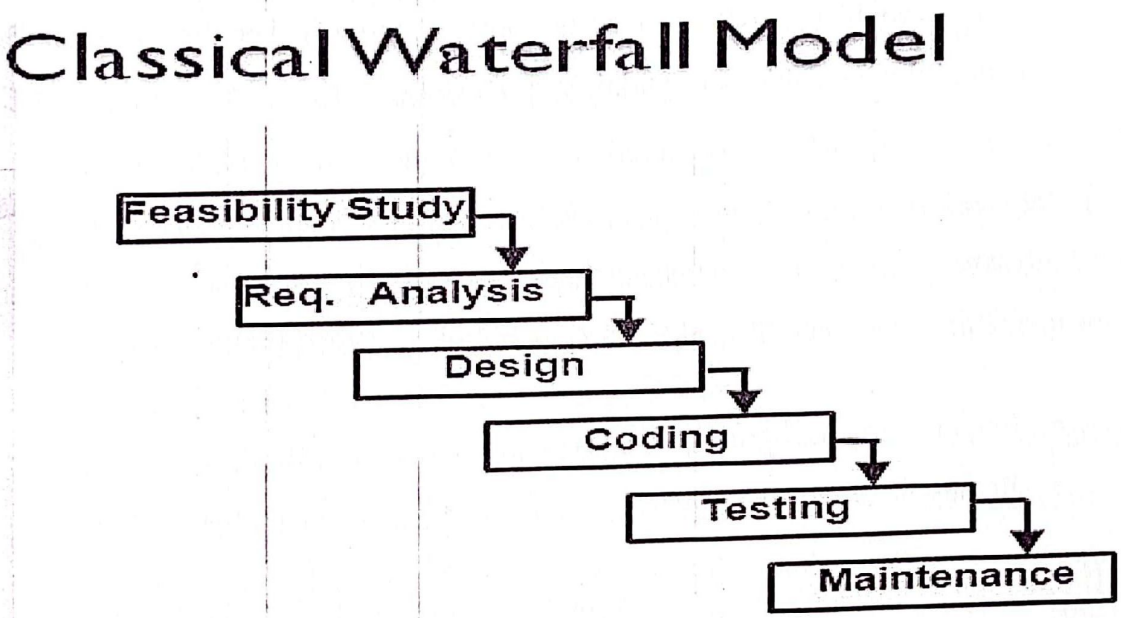


Figure 1c: Waterfall Model.

i) **Feasibility study** - The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.

- At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system.

- After they have an overall understanding of the problem they investigate the different solutions that are possible. Then they examine each of the solutions in

terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution.

- Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

ii) Requirements analysis and specification: The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis
- Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed. The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements.

The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start.

During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation

iii) **Design: -** The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

a) **Traditional design approach** -Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.

b) **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.

iv. **Coding and unit testing:-**The purpose of the coding phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested. During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves

testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

v) **Integration and system testing:** -Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

i. α – testing: It is the system testing performed by the development team.

ii. β –testing: It is the system testing performed by a friendly set of customers.

iii. Acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed, specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.

vi) **Maintenance:** -Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60

ratios. Maintenance involves performing any one or more of the following three kinds of activities:

i. Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.

ii. Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called perfective maintenance.

iii. Adapting 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. This is called adaptive maintenance.

Shortcomings of the classical waterfall model

The classical waterfall model is an idealistic one since it assumes that no development error is ever committed by the engineers during any of the life cycle phases. However, in practical development environments, the engineers do commit a large number of errors in almost every phase of the life cycle. The source of the defects can be many: oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc. These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till we reach the coding or testing phase. Once a defect is detected, the engineers need to go back to the phase where the defect had occurred and redo some of the work done during that phase and the subsequent phases to correct the defect and its effect on the later phases. Therefore, in any practical software development work, it is not possible to strictly follow the classical waterfall model.

2. ITERATIVE WATERFALL MODEL

The introduction of iterative waterfall model was to overcome the major shortcomings of the classical waterfall model. Here, a feedback is provided for error correction as & when detected later in a phase. Though errors are inevitable, but it is desirable to detect them in the same phase in which they occur. If so, this can reduce the effort to correct the bug.

The advantage of this model is that, there is a working model of the system at a very early stage of development which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.

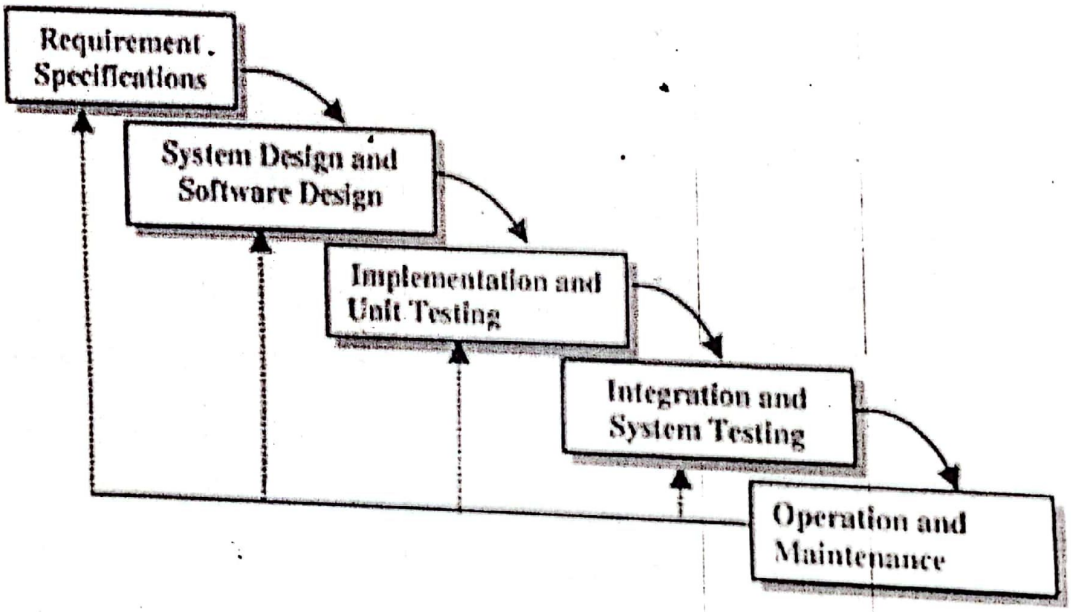


Figure 1d: Iterative Waterfall Model

The disadvantage with this SDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

3. PROTOTYPE MODEL

Prototype

A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations. A prototype usually turns out to be a very crude version of the actual system.

Need for a prototype in software development

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs:

- how the screens might look like
- how the user interface would behave
- how the system would produce outputs

Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.

A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues

associated with the product development. Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.

A prototype of the actual product is preferred in situations such as:

- User requirements are not complete
- Technical issues are not clear

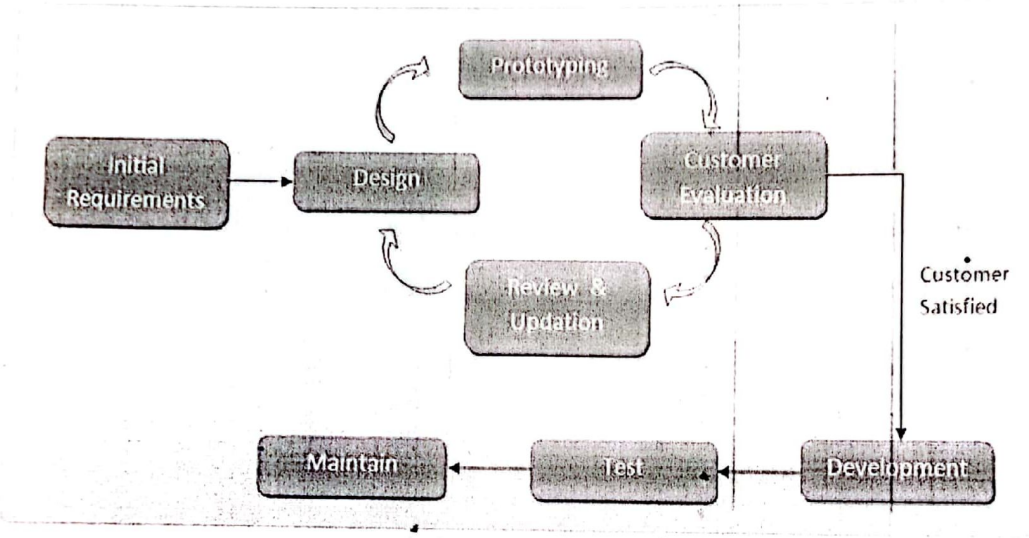


Fig 1.6 Prototype Model

4. EVOLUTIONARY MODEL

It is also called *successive versions model* or *incremental model*. At first, a simple working model is built. Subsequently it undergoes functional improvements & we keep on adding new functions till the desired system is built.

Applications:

- Large projects where you can easily find modules for incremental implementation.
- Often used when the customer wants to start using the core features rather than waiting for the full software.

- Also used in object oriented software development because the system can be easily portioned into units in terms of objects.

Advantages:

i. User gets a chance to experiment partially developed system

ii. Reduce the error because the core modules get tested thoroughly.

Disadvantages:

i. It is difficult to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented & delivered

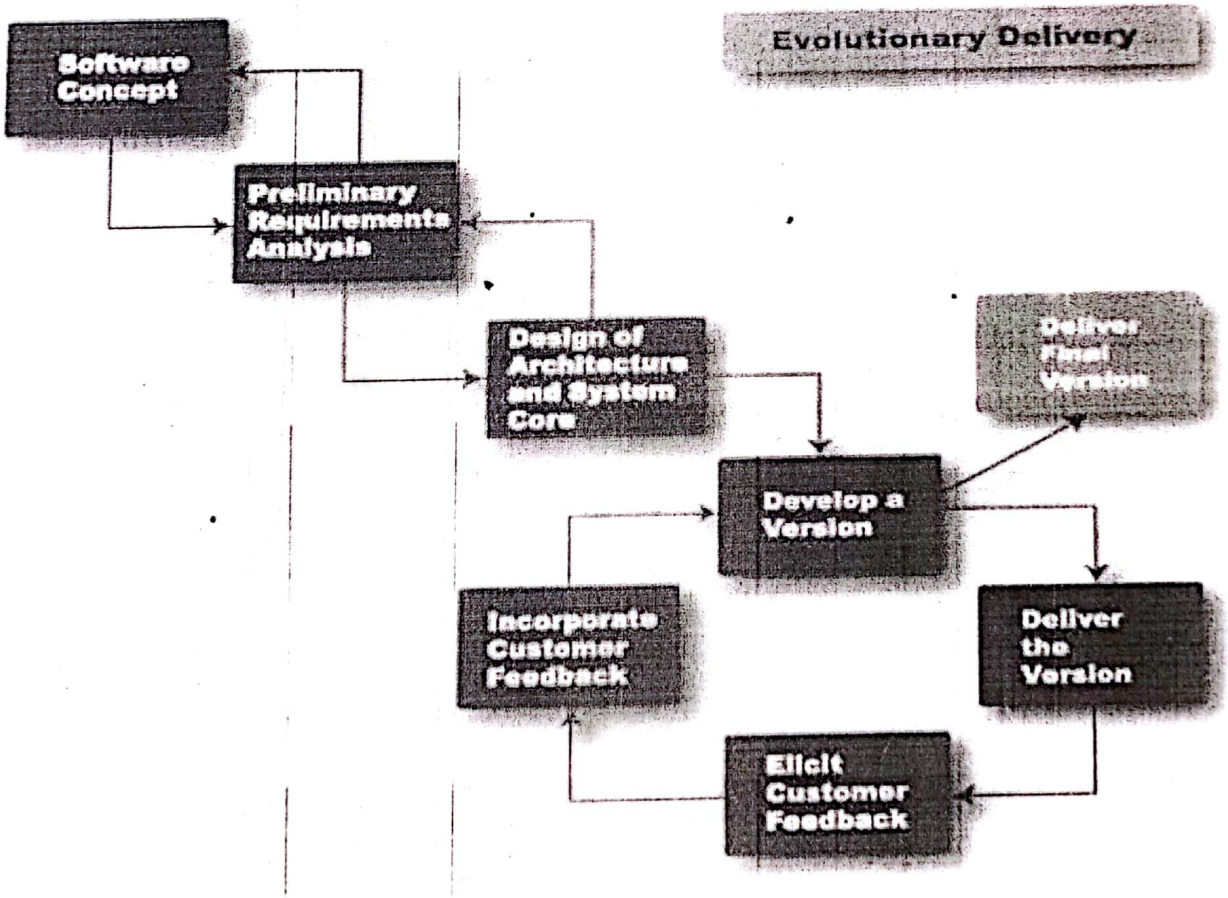


Fig 1d Evolutionary model.

5. SPIRAL MODEL

The Spiral model of software development is shown in fig. 1e. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study, the next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig. 1e. The following activities are carried out during each phase of a spiral model.

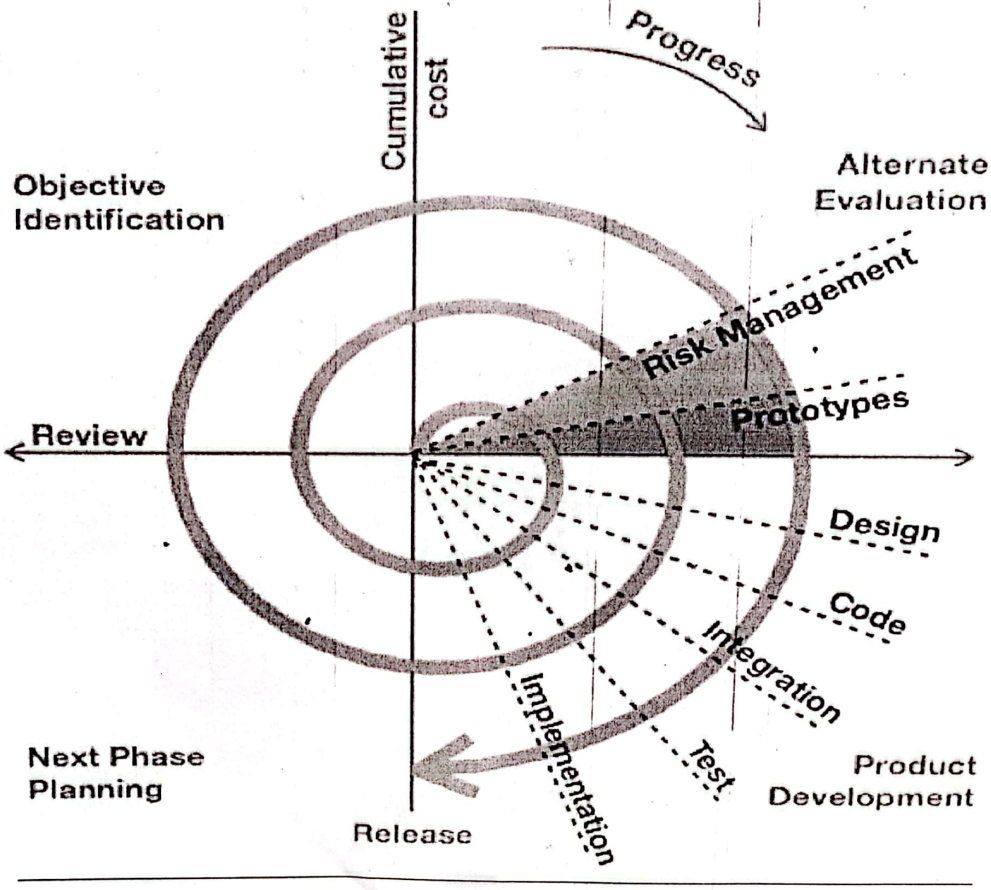


Fig 1e: Spiral model.

First quadrant (Objective Setting)

- During the first quadrant, it is needed to identify the objectives of the phase.
- Examine the risks associated with these objectives.

Second Quadrant (Risk Assessment and Reduction)

- A detailed analysis is carried out for each identified project risk.
- Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

Third Quadrant (Development and Validation)

- Develop and validate the next level of the product after resolving the identified risks.

Fourth Quadrant (Review and Planning)

- Review the results achieved so far with the customer and plan the next iteration around the spiral.
- Progressively more complete version of the software gets built with each iteration around the spiral.

Circumstances to use spiral model

The spiral model is called a meta model since it encompasses all other development process models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models, this is probably a factor deterring its use in ordinary projects.

4. Project Scheduling, Planning and Tracking

Schedule is another important factor in many projects. Business trends are dictating that the time to market of a product should be reduced; that is, the cycle time from concept to delivery should be small. For software this means that it needs to be developed faster, and within the specified time.

Unfortunately, the history of software is full of cases where projects have been substantially late. Clearly, therefore, reducing the cost and the cycle time for software development are central goals of software engineering.

Project Scheduling refers to a roadmap of all activities to be done with specified order and within the time slot allotted to each activity. Project managers tend to define various tasks, and project milestones and then arrange them keeping various factors in mind. They look for tasks like in critical path in the schedule, which are necessary to complete in specific manner (because of task interdependency) and strictly within the time allocated. Arrangement of tasks which lies out of critical path are less likely to impact over all schedule of the project.

For scheduling a project, it is necessary to -

- Break down the project tasks into smaller, manageable form
- Find out various tasks and correlate them
- Estimate time frame required for each task
- Divide time into work-units
- Assign adequate number of work-units for each task
- Calculate total time required for the project from start to finish

Planning is the most important project management activity. It has two basic objectives—establish reasonable cost, schedule, and quality goals for the project, and to draw out a plan to deliver the project goals. A project succeeds if it meets its cost, schedule, and quality goals. Without the project goals being defined, it is not possible to even declare if a project has succeeded. And without detailed planning, no real monitoring or controlling of the project is possible. Often projects are rushed toward implementation

with not enough effort spent on planning. No amount of technical effort later can compensate for lack of careful planning. Lack of proper planning is a sure ticket to failure for a large software project. For this reason, we treat project planning as an independent chapter. Note that we also cover the monitoring phase of the project management process as part of planning, as how the project is to be monitored is also a part of the planning phase.

The inputs to the planning activity are the requirements specification and maybe the architecture description. A very detailed requirements document is not essential for planning, but for a good plan all the important requirements must be known, and it is highly desirable that key architecture decisions have been taken.

There are generally two main outputs of the planning activity: the overall project management plan document that establishes the project goals on the cost, schedule, and quality fronts, and defines the plans for managing risk, monitoring the project, etc.; and the detailed plan, often referred to as the detailed project schedule, specifying the tasks that need to be performed to meet the goals, the resources who will perform them, and their schedule. Overall plan guides the development of the detailed plan, which then becomes the main guiding document during project execution for project monitoring.

Effort Estimation

For a software development project, overall effort and schedule estimates are essential prerequisites for planning the project. These estimates are needed before development is initiated, as they establish the cost and schedule goals of the project. Without these, even simple questions like "is the project late?" "are there cost overruns?" and "when is the project likely to complete?" cannot be answered. A more practical use of these estimates is in bidding for software projects, where cost and schedule estimates must be given to a potential client for the development contract. (As the bulk of the cost of software development is due to the human effort, cost can easily be determined from effort by using a suitable person-month cost value.) Effort and schedule estimates are also required

for determining the staffing level for a project during different phases, for the detailed plan, and for project monitoring.

The accuracy with which effort can be estimated clearly depends on the level of information available about the project. The more detailed the information, the more accurate the estimation can be.

Quality Planning

Having set the goals for effort and schedule, the goal for the third key dimension of a project—quality—needs to be defined. However, unlike schedule and effort, quantified quality goal setting for a project and then planning to meet it is much harder. For effort and schedule goals, we can easily check if a detailed plan meets these goals (e.g., by seeing if the last task ends before the target date and if the sum total of effort of all tasks is less than the overall effort goal). For quality, even if we set the goal in terms of expected delivered defect density, it is not easy to plan for achieving this goal or for checking if a plan can meet these goals. Hence, often, quality goals are specified in terms of acceptance criteria—the delivered software should finally work for all the situations and test cases in the acceptance criteria. Further, there may even be an acceptance criterion on the number of defects that can be found during the acceptance testing. For example, no more than n defects are uncovered by acceptance testing.

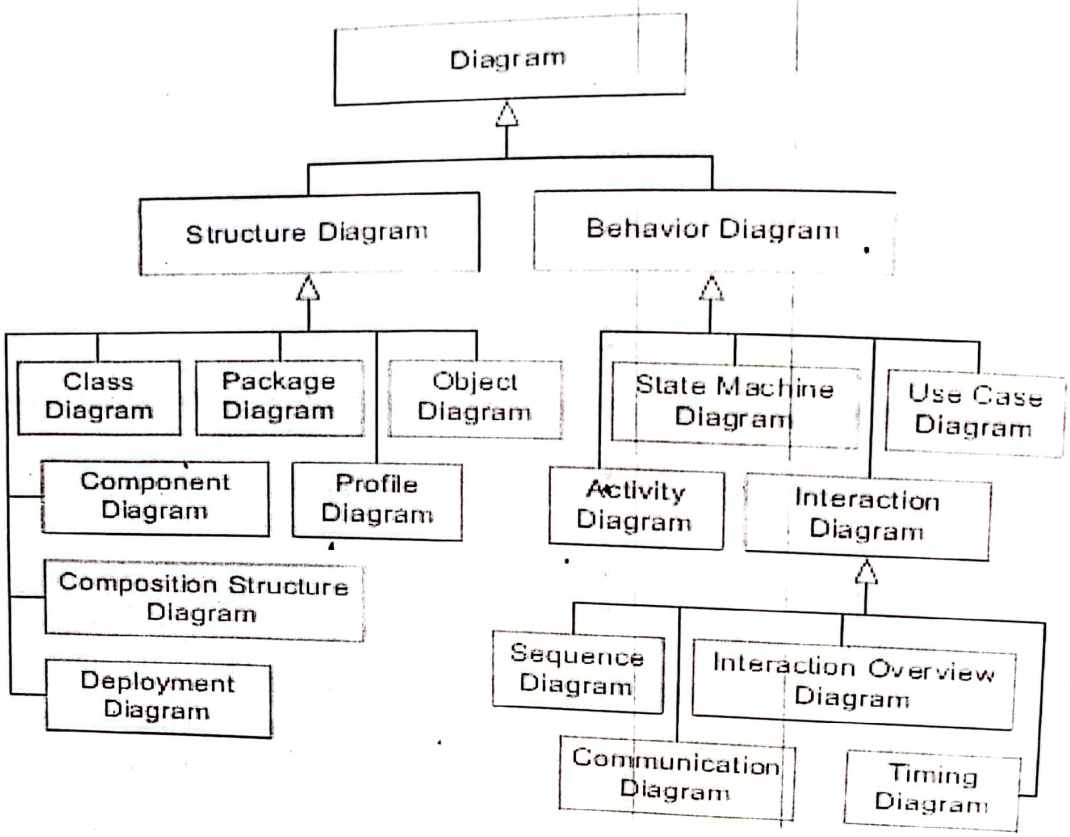
The quality plan is the set of quality-related activities that a project plans to do to achieve the quality goal. To plan for quality, let us first understand the defect injection and removal cycle, as it is defects that determine the quality of the final delivered software. Software development is a highly people-oriented activity and hence it is error-prone. In a software project, we start with no defects (there is no software to contain defects).

Defects are injected into the software being built during the different phases in the project. That is, during the transformation from user needs to software to satisfy those needs, defects are injected in the transformation activities undertaken. These injection stages are primarily the requirements specification, the high-level design, the detailed design, and coding. To ensure that high-quality software is delivered, these defects are removed through the quality control (QC) activities. The QC activities for defect removal include requirements reviews, design reviews, code reviews, unit testing, integration testing, system testing, acceptance testing, etc.

5. Unified Modelling Language

UML is a graphical notation for expressing object-oriented designs. It is called a modeling language and not a design notation as it allows representation of various aspects of the system, not just the design that has to be implemented. For an OO design, a specification of the classes that exist in the system might suffice. However, while modeling, during the design process, the designer also tries to understand how the different classes are related and how they interact to provide the desired functionality. This aspect of modeling helps to build the designs that are more likely to satisfy the requirements of the system.

In UML, a model is represented graphically in the form of *diagrams*. A *Diagram* provides a view of that part of reality described by the model. There are diagrams that express which users use which functionality and diagrams that show the structure of the system but without specifying a concrete implementation. Most recent UML tools offers at least 14 diagrams that describe either the structure or the behavior of a system.



A chart showing a taxonomy of the UML diagrams

UML offers seven types of diagrams for modeling the structure of a system from different perspectives. The dynamic behavior of the elements in question (i.e., their changes over time) is not considered in these diagrams. Some of the diagrams that are commonly used are briefly discussed.

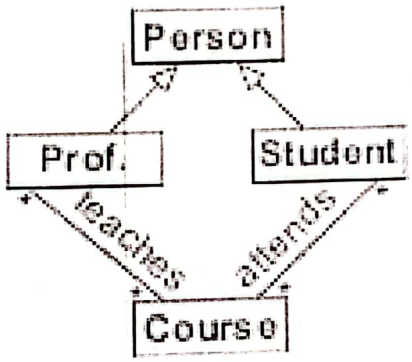
A.) *Structured Diagrams*

UML offers several structured diagrams for modeling the structure of a system from different perspectives as shown in the diagram above. The dynamic behavior of the elements in question (i.e., their changes over time) is not considered in these diagrams.

1. The class diagram

Just like the concepts of the object diagram, the concepts of the *class diagram* originate from conceptual data modeling and object-oriented software development. These

concepts are used to specify the data structures and object structures of a system. The class diagram is based primarily on the concepts of *class*, *generalization*, and *association*. For example, in a class diagram, you can model the classes Course, Student, and Professor in a system. Professors teach courses and students attend courses. Students and professors have common properties as they are both members of the class Person. This is expressed by a generalization relationship.



The class diagram of UML is the central piece in a design or model. As the name suggests, these diagrams describe the classes that are there in the design. As the final code of an OO implementation is mostly classes, these diagrams have a very close relationship with the final code. There are many tools that translate the class diagrams to code skeletons, thereby avoiding errors that might get introduced if the class diagrams are manually translated to class definitions by programmers. A class diagram defines:

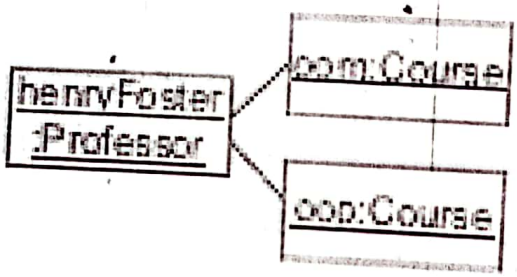
i. Classes that exist in the system—besides the class name, the diagrams are capable of describing the key fields as well as the important methods of the classes.

ii. Associations between classes—what types of associations exist between different classes.

iii. Subtype, super-type relationship—classes may also form subtypes giving type hierarchies using polymorphism. The class diagrams can represent these hierarchies also.

2. The Object Diagram

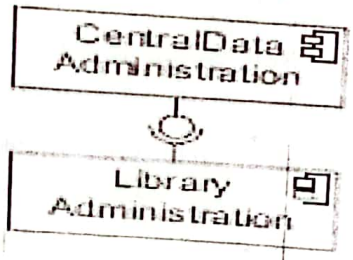
Based on the definitions of the related class diagram, an *object diagram* shows a concrete snapshot of the system state at a specific execution time. For example, an object diagram could show that a professor Henry Foster (henryFoster) teaches the courses Object-Oriented Modeling (OOM) and Object-Oriented Programming (OOP).



An object diagram

3. The Component Diagram

UML pays homage to component-oriented software development by offering *component diagrams*. A component is an independent, executable unit that provides other components with services or uses the services of other components. UML does not prescribe any strict separation between object-oriented and component-oriented concepts. Indeed, these concepts may be combined in any way required. When specifying a component, you can model two views explicitly: the external view (black box view), which represents the specification of the component, and the internal view (white box view), which defines the implementation of the component.

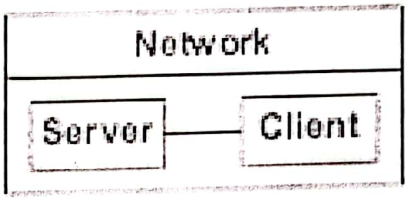


Component diagram

4. The Composition Diagram

The *composition structure diagram* allows a hierarchical decomposition of the part of the system. You can therefore use a composition structure diagram to describe the internal structure of classes or components in detail. This enables you to achieve a higher level of detail than, for example, in a class diagram because the modeling is context-specific.

The details of the internal structure that are valid precisely for the context under consideration can be specified.



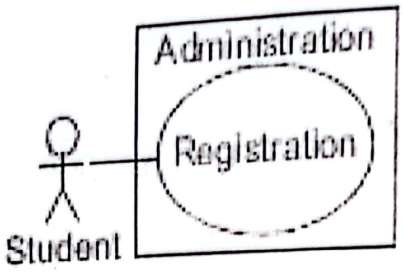
The Composition structure diagram

B.) Behavior Diagrams

With the *behavior diagrams*, UML offers the infrastructure that enables you to define behavior in detail. Behavior refers to the direct consequences of an action of at least one object. It affects how the states of objects change over time. Behavior can either be specified through the actions of a single object or result from interactions between multiple objects.

1. The Use Case Diagram

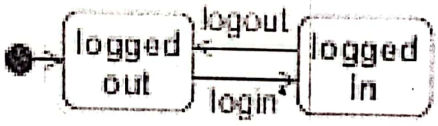
UML offers the *use case diagram* to enable you to define the requirements that a system must fulfill. This diagram describes which users use which functionalities of the system but does not address specific details of the implementation. The units of functionality that the system provides for its users are called *use cases*. In a university administration system, for example, the functionality Registration would be a use case used by students.



Use case diagram

2. The State Machine Diagram

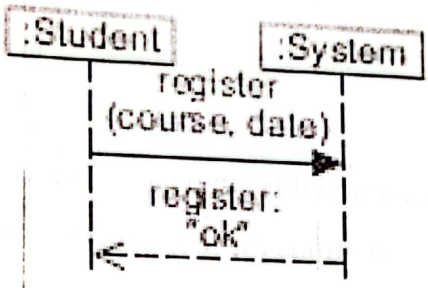
Within their life cycle, objects go through different states. For example, a person is in the state logged out when first visiting a website. The state changes to logged in after the person successfully entered username and password (event login). As soon as the person logs out (event logout), the person returns to the state logged out. This behavior can be represented in UML using the *state machine diagram*. This diagram describes the permissible behavior of an object in the form of possible states and state transitions triggered by various events.



State machine diagram

3. The Sequence Diagram

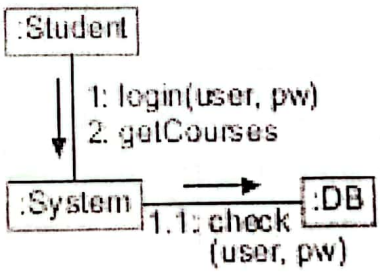
The *sequence diagram* describes the interactions between objects to fulfill a specific task, for example, registration for an exam in a university administration system. The focus is on the chronological order of the messages exchanged between the interaction partners. Various constructs for controlling the chronological order of the messages as well as concepts for modularization allow you to model complex interactions.



Sequence diagram

4. Communication diagram

Similarly to the sequence diagram, the *communication diagram* describes the communication between different objects. Here, the focus is on the communication relationships between the interaction partners rather than on the chronological order of the message exchange. Complex control structures are not available. This diagram clearly shows who interacts with whom.



Communication diagram

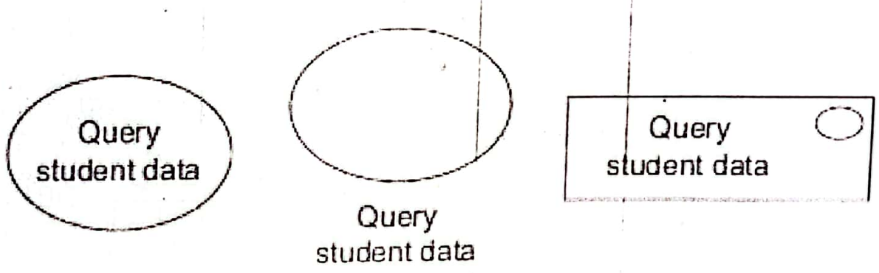
Further emphasis on use case, sequence diagram and Activity diagram

i. USE CASE

A *use case* describes functionality expected from the system to be developed. It encompasses a number of functions that are executed when using this system. A use case provides a tangible benefit for one or more actors that communicate with this use case. The use case diagram does not cover the internal structure and the actual implementation of a use case.

Use cases are determined by collecting customer wishes and analyzing problems specified in natural language when these are the basis for the requirements analysis. However, use cases can also be used to document the functionality that a system offers. A use case is usually represented as an ellipse. The name of the use case is specified directly in or directly beneath the ellipse. Alternatively, a use case can be represented by a rectangle that contains the name of the use case in the center and a small ellipse in the top right-hand corner.

Notation alternatives for use cases:



The set of all use cases together describes the functionality that a software system provides. The use cases are generally grouped within a rectangle. This rectangle symbolizes the boundaries of the *system* to be *System* described.

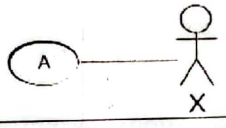
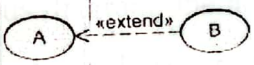
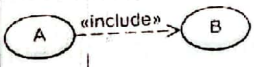
The example in figure below shows the Student Administration system, which offers three use cases:

(1) Query student data,

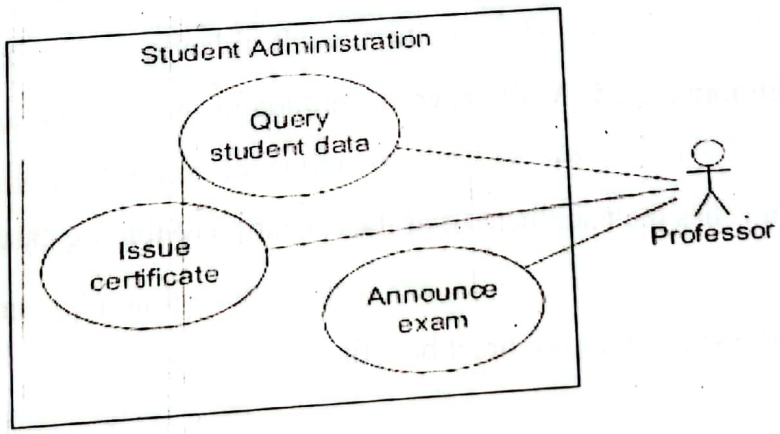
(2) Issue certificate, and

(3) Announce exam.

Notation elements:

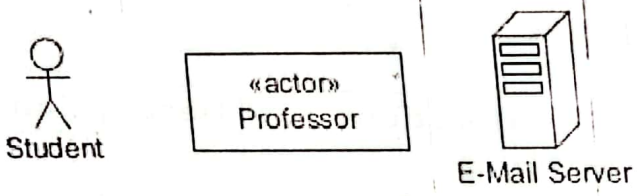
Name	Notation	Description
Association		Relationship between use cases and actors
Generalization		Inheritance relationship between actors or use cases
Extend relationship		B extends A: optional use of use case B by use case A
Include relationship		A includes B: required use of use case B by use case A

These use cases may be triggered by the actor Professor.



Actors

To describe a system completely, it is essential to document not only what the system can do but also who actually works and interacts with the system. In the use case diagram, *actors* always interact with the term in the context of their use cases, that is, the use cases with which they are associated. These three notation alternatives:



are all equally valid. As we can see from this example, actors can be *human* (e.g., student or professor) or *non-human* (e.g., e-mail server). The symbols used to represent the actors in a specific use case diagram depend on the person creating the model or the tool used.

Types of actors:

- *Human/non-human*
- *Active/passive*
- *Primary/secondary*

1. **Human** E.g., **Student**, **Professor**

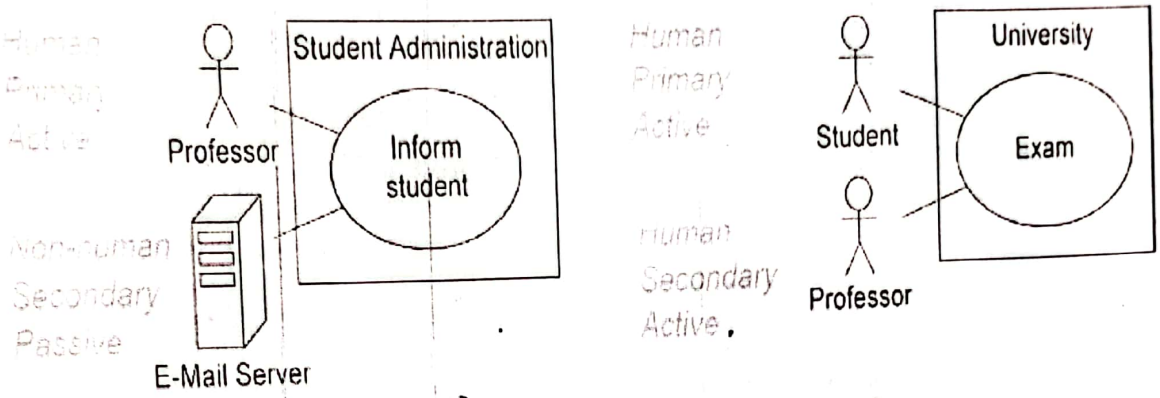
2. **Non-human** E.g., **E-Mail Server**

3. **Primary**: has the main benefit of the execution of the use case

4. **Secondary**: receives no direct benefit

5.Active: initiates the execution of the use case

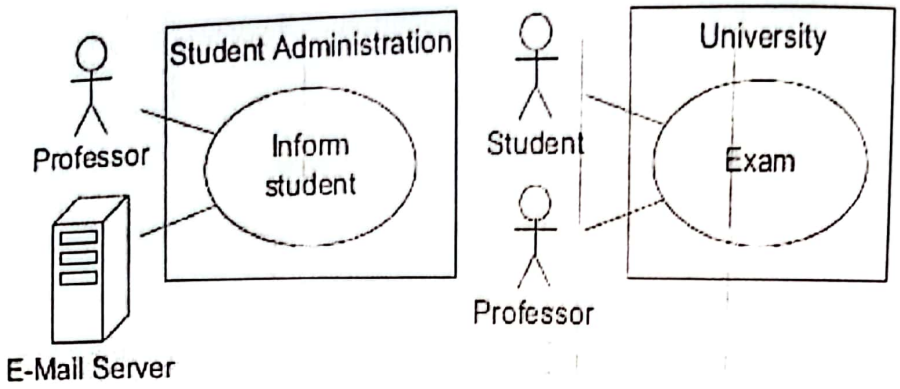
6.Passive: provides functionality for the execution of the use case



An actor interacts with the system by using the system as an active actor, meaning that the actor is a *passive* actor providing functionality for the execution of use cases.

In the Immediate previous diagram, the actor Professor is an active actor, whereas the actor E-Mail Server is passive. However, both are required for the execution of the use case Inform student. Furthermore, use case diagrams can also contain both *primary* and *secondary* actors, also shown in this example. A primary actor takes an actual benefit from the execution of the use case (in our example, this is the Professor), whereas the secondary actor E-Mail Server receives no direct benefit from the execution of the use case. The secondary actor does not necessarily have to be passive. Both the Professor and the Student are actively involved in the execution of the use case Exam, whereby the main beneficiary is the Student. In contrast, the Professor has a lower benefit from the exam but is necessary for the execution of the use case. Graphically, there is no differentiation between primary and secondary actors, between active and passive actors, and between human and non-human actors.

Examples of actors

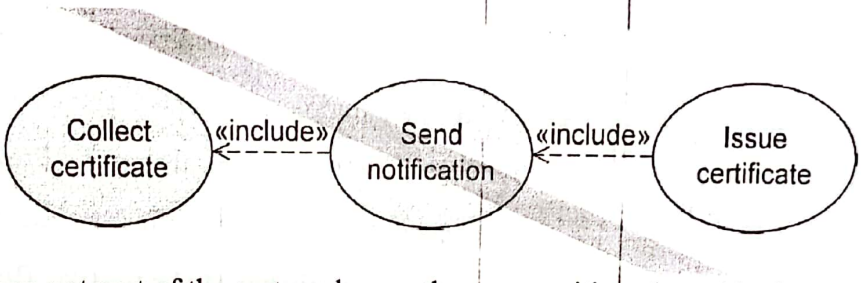


(a) (b)

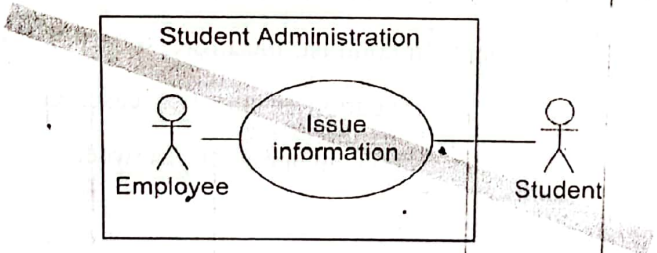
An actor is always clearly outside the system, i.e., a user is never part of the system and is therefore never implemented.

Typical Errors To Avoid

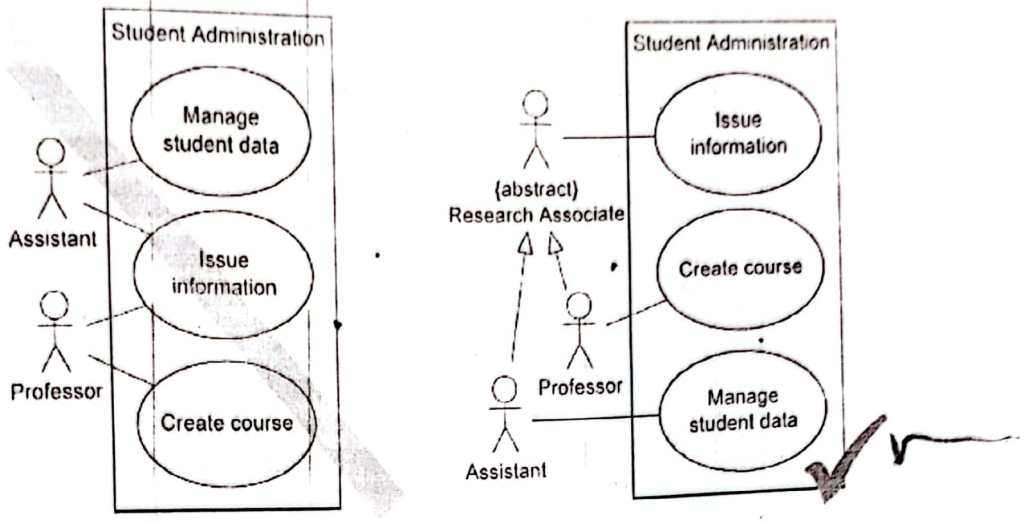
1. Use case diagrams do not model processes/workflows!



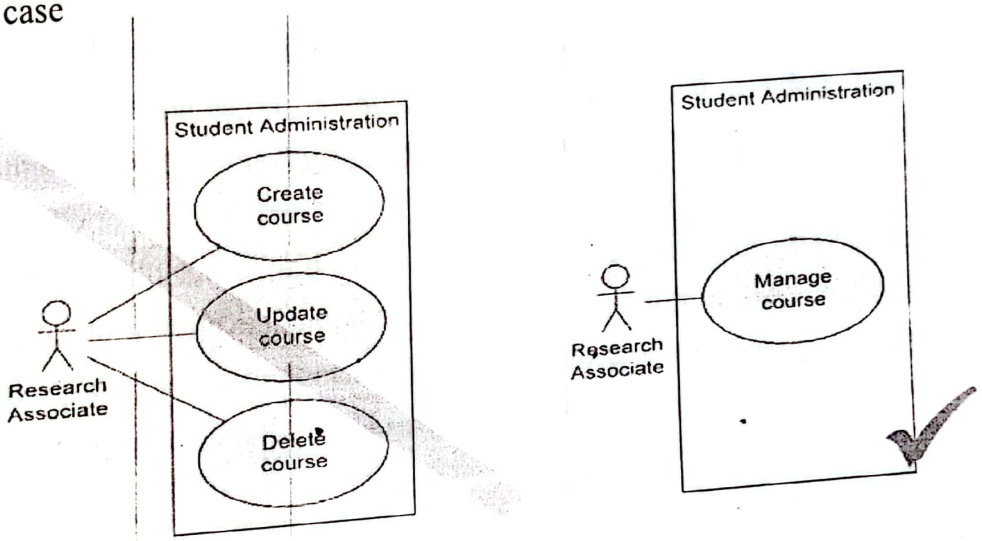
2. Actors are not part of the system, hence, they are positioned outside the system boundaries!



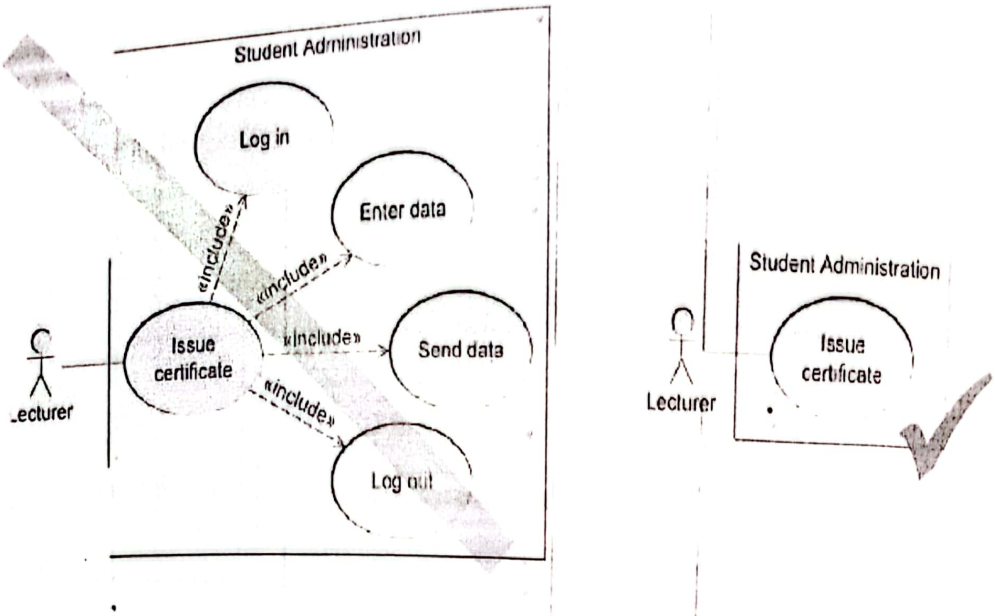
3. Use case Issue information needs to one unique actor for execution.



4. Many small use cases that have the same objective may be grouped to form one use case



5. The various steps are part of the use cases, not separate use cases themselves! -> NO functional decomposition



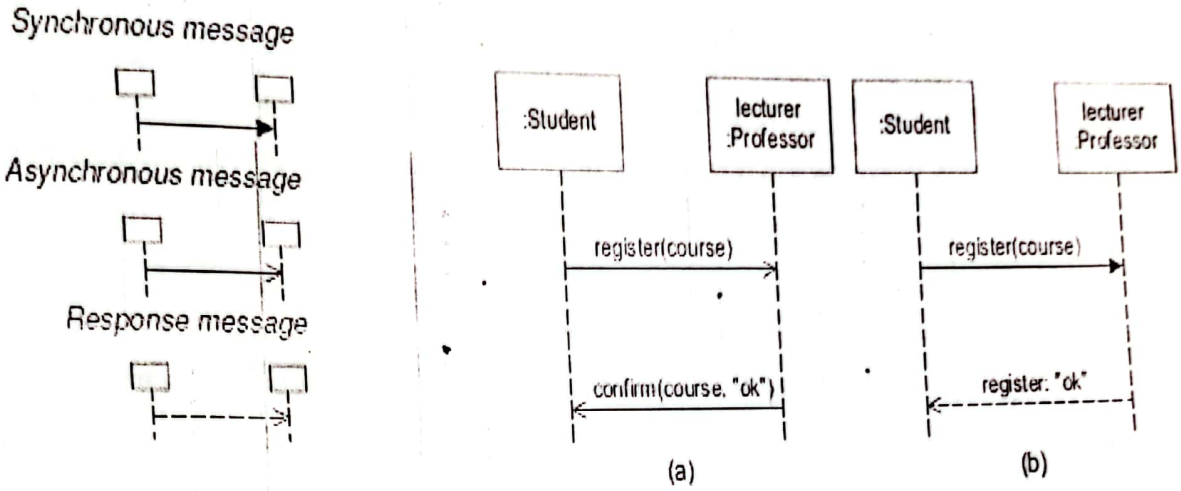
ii. Sequence diagram

This diagram looks at the modeling of the *inter*-object behavior, i.e., the interactions between the objects in a system. An *interaction* specifies how messages and data are exchanged between interaction partners. The *interaction partners* are either human such as lecturers or students, or non-human, such as a server, a printer, or executable software. An interaction can be a conversation between multiple persons, for instance, an oral exam. Alternatively, an interaction can model communication protocols such as HTTP or represent the message exchange between humans and a software system.

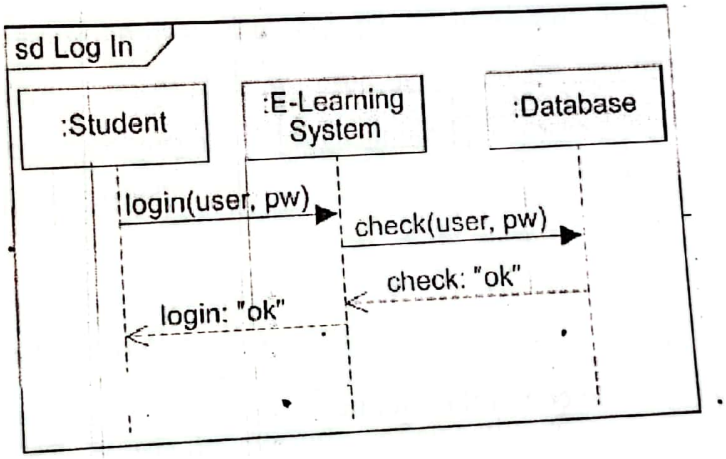
Messages

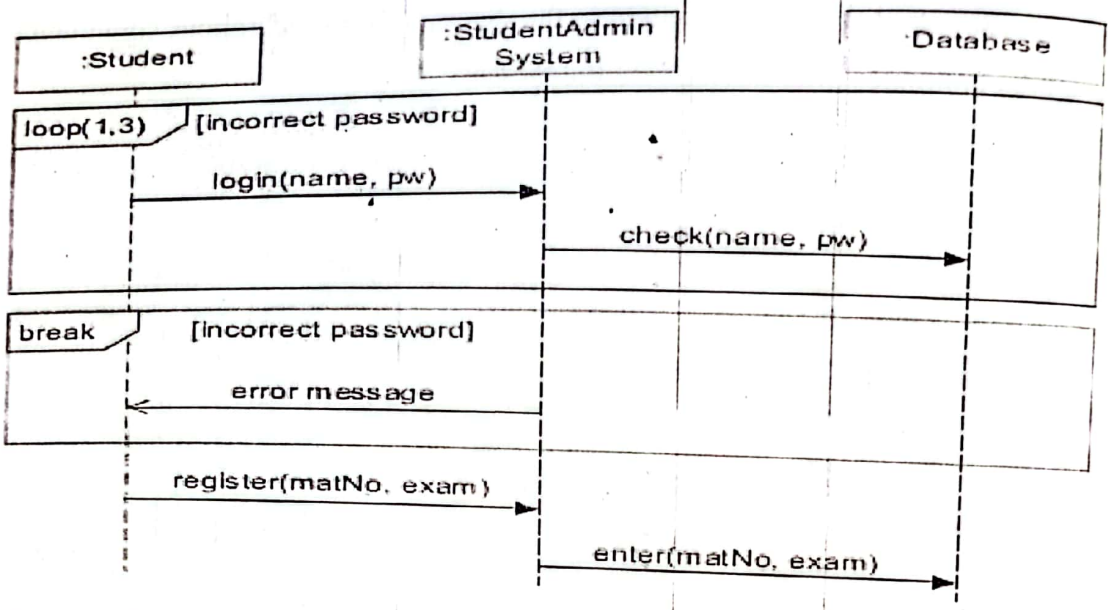
In a sequence diagram, a *message* is depicted as an arrow from the sender to the receiver. The type of the arrow expresses the type of communication involved. A *synchronous message* is represented by an arrow with a continuous line and a filled triangular arrowhead. An *asynchronous message* is depicted by an arrow with a continuous line and an open arrowhead. In the case of synchronous messages, the sender waits until it has received a *response message* before continuing. The response message is represented by a dashed line with an open arrowhead. If the content of the response message and the point at which the response message is sent and received are clear from the context, then

the response message may be omitted in the diagram. In asynchronous communication, the sender continues after having sent the message.



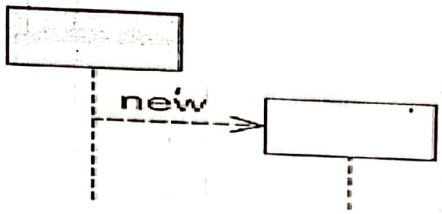
More illustrations:





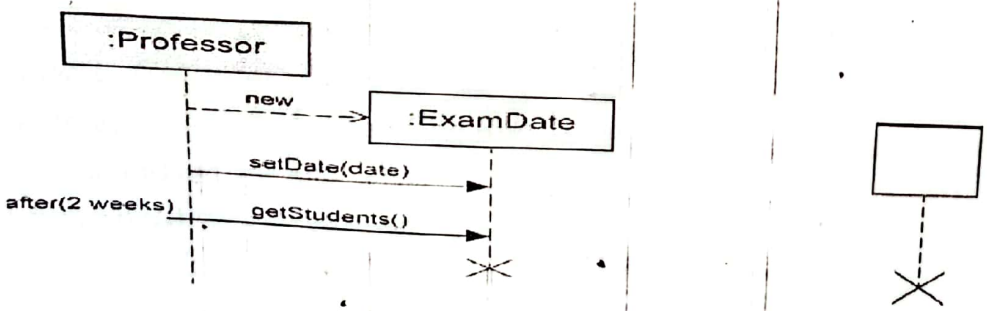
Object creation

Dashed arrow
Arrowhead points to the head of the lifeline of the object to be created
Keyword **new**



Object destruction

Object is deleted
Large cross (×) at the end of the lifeline



iii. *Activity diagram*

The *activity diagram* focuses on modeling procedural processing aspects of a system. It specifies the control flow and data flow between various steps, the *actions* required to implement an activity. Activity diagrams are also based on established concepts for describing concurrent communicating processes. An activity diagram allows you to specify user-defined behavior in the form of activities. An *activity* itself can describe the implementation of a use case.

The basic elements of activities are *actions*. Just like an activity, an action is depicted as a rectangle with rounded corners, whereby the name of the action is positioned centrally within the rounded rectangle. You can use actions to specify any user-defined behavior.

The UML standard does not stipulate any specific form of notation for activities. In addition to the flow-based notation elements of the activity diagrams, the standard also allows other forms of notation, such as structural diagrams or even pseudocode. A number of recurring control flow and data flow patterns have emerged in addition to custom notation elements. They are used in particular for modeling business processes and have proven to be very useful for complex processes. These constructs are referred to as "workflow patterns".

Beginning and Termination of Activities

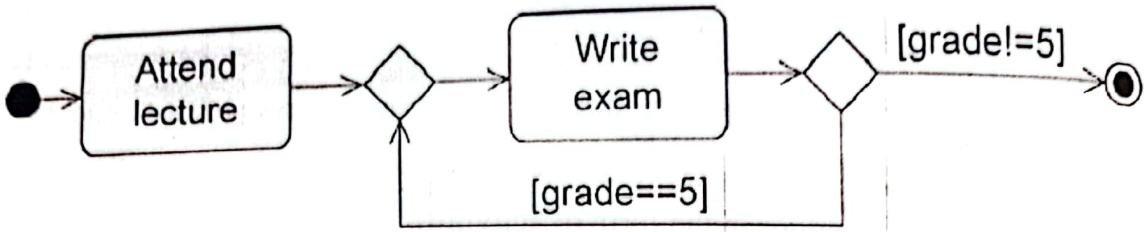
● Initial node : Starts the execution of an activity

◉ Activity final node : ~~Starts~~ End the execution of an activity

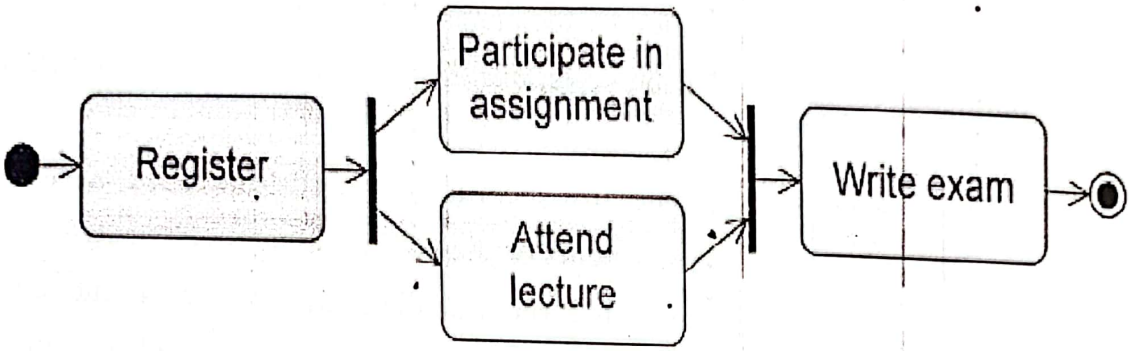
⊗ Flow final node : Ends one execution path of an activity

Decision and merge nodes can also be used to model loops:

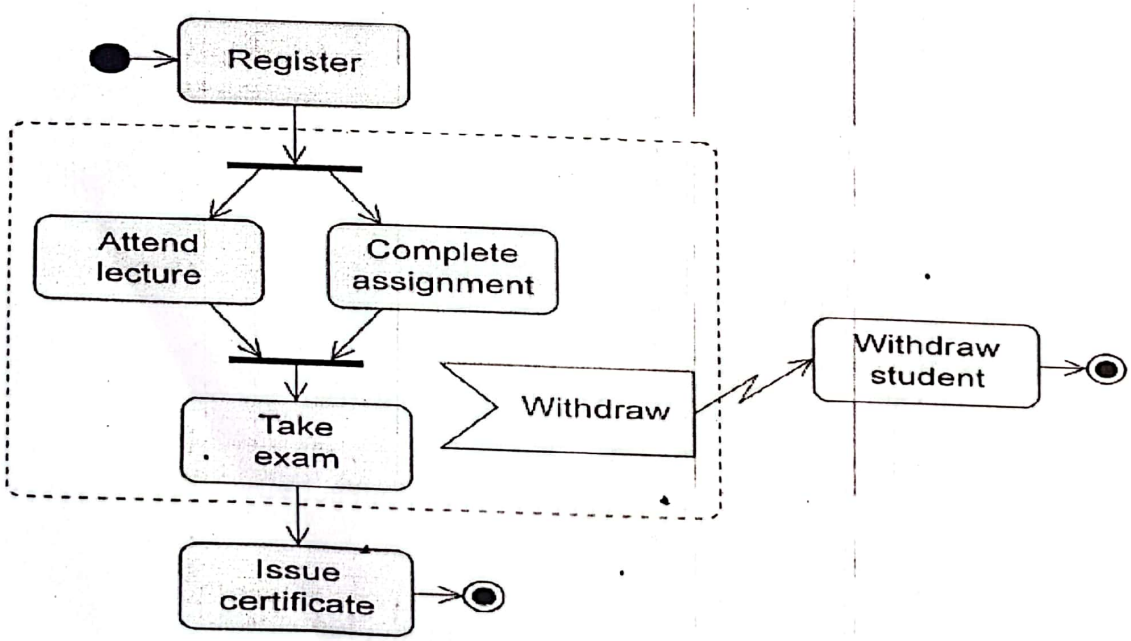
Example 1



Example 2



Example 3



6. SOFTWARE TESTING

Software testing is the process of finding errors and validating the software/system against its specifications. Testing software typically involves:

- Executing software with inputs representative of actual operation conditions (or operational profiles),
- Comparing the product/expected outputs
- Measuring execution characteristics (e.g., memory used, time consumed, etc.)

Testing, therefore, measures software quality and it is capable of finding faults. When faults are removed, software quality and possibly reliability is improved. Testers should note that exhaustive testing is impossible, this implies that exercising all combinations of inputs and preconditions is not realistic.

Terminologies

Fault: an imperfection that may lead to a failure •E.g., missing/incorrect code that may result in a failure
Bug: another name for a fault in code
Error: where the system state is incorrect but may not have been observed
Test Case: set of inputs, execution conditions, and expected results developed for a particular objective
Test Suite: collection of test cases, typically related by a testing goal or an implementation dependency
Test Strategy: algorithm or heuristic to create test cases from a representation, implementation, or a test model
Oracle: means to check the output from a program is correct for the given input

Testing scope

- **"Testing in the small"(unit testing)**

Exercising the smallest executable units of the system

- **"Testing the build"(integration testing)**

•Finding problems in the interaction between components

- **"Testing in the large"(system testing)**

Putting the entire system to test

General Testing Principles

1.Testing shows presence of Defects
2.Exhaustive Testing is Impossible!
3.Early Testing
4.Defect Clustering
5.The Pesticide Paradox
6.Testing is Context Dependent
7. Absence of Errors Fallacy

1. **Testing shows the presence of Defects**

We test to find Faults (a.k.a Defects): As we find more defects, the probability of undiscovered defects remaining in a system reduces. However Testing cannot prove that there are no defects present.

2. **Exhaustive Testing is Impossible!**

It is a clear fact that we cannot test everything (i.e. all combinations of inputs and pre-conditions). Therefore, we must Prioritise our testing effort using a Risk Based Approach.

3. **Early testing**

Testing activities should start as early as possible in the development life cycle. These activities should be focused on defined objectives outlined in the Test Strategy.

4. Defect Clustering

Defects are not evenly spread in a system, they are 'clustered'. In other words, most defects found during testing are usually confined to a small number of modules. Similarly, most operational failures of a system are usually confined to a small number of modules, an important consideration in test prioritization!

5. The Pesticide Paradox

Testing identifies bugs, and programmers respond to fix them and as bugs are eliminated by the programmers, the software improves. As software improves the effectiveness of previous tests
erodes. Therefore, we must learn, create and use new tests based on new techniques to catch new bugs.

> ***Note:*** *It's called the "pesticide paradox" after the agricultural phenomenon, where bugs such as the boll weevil build up tolerance to pesticides, leaving you with the choice of evermore powerful pesticides followed by ever-more powerful bugs or an altogether different approach.' – Beizer 1995*

7. Testing is Context Dependent

• Testing is done differently in different contexts, for instance, safety-critical software is tested differently from an e-commerce site. At times, 3 to 10 failures per thousand lines of code is typical for commercial software and 1 to 3 failure is typical for industrial software. Also different industries impose different testing standards.

8. Absence of Errors Fallacy

• If we build a system and, in doing so, find and fix defects, it doesn't make it a good system even after defects have been resolved, it may still be unusable and/or does not fulfil the users' needs and expectations.

Categories of test case design techniques.

Equivalence partitioning
Boundary value analysis
Decision table testing
State transition testing
Use case testing

Due to constraints, our focus will be on equivalent partitioning and boundary value analysis as they appear to be the most important.

Black Box vs White Box Testing

Historically, testing techniques have been distinguished either as black box or white box. Black-box techniques (which include behavioral or specification-based techniques), permits the selection of test conditions or test cases based on an analysis of software design documentation, whether functional or non-functional, for a component or system without reference to its internal structure (i.e. in the absence of source codes). White box techniques (also called structural or structure-based techniques) permits the selection of test cases based on the dynamic analysis of the structure of the software system under test via its execution. Thus, in some cases, there may be a situation where some kind of instrumentation may be required in the code either manually or automatically (by an automation tool) to permit such an analysis. The common features for black box and white box testing techniques can be summarized as shown in the following table:

Common Features for Black Box and White Box Testing Techniques

Black Box Testing	White Box Testing
• Models, either formal or informal, are used for the specification of the problem to be solved, the software or its components.	• Information about how the software is constructed is used to derive the test cases, for example, code and design.
• From these models test cases can be derived systematically.	• The extent of coverage of the software can be measured for existing test cases, and further test cases can be derived systematically to increase coverage.

Equivalence partitioning

Input partitions

Equivalence partitioning is based on a very simple idea: it is that in many cases the inputs to a program can be 'chunked' into groups of similar inputs. For example, a program that accepts integer values can accept as valid any input that is an integer (i.e. a whole number) and should reject anything else (such as a real number or a character). The range of integers is infinite, though the computer will limit this to some finite value in both the

negative and positive directions (simply because it can only handle numbers of a certain size; it is a finite machine).

Let us suppose, for the sake of an example, that the program accepts any value between −10,000 and +10,000 (computers actually represent number in binary form, which makes the numbers look much less like the ones we are familiar with, but we will stick to a familiar representation). If we imagine a program that separates numbers into two groups according to whether they are positive or negative the total range of integers could be split into three 'partitions': the values that are less than zero; zero; and the values that are greater than zero. Each of these is known as an 'equivalence partition' because every value inside the partition is exactly equivalent to any other value as far as our program is concerned.

So if the computer accepts −2,905 as a valid negative integer we would expect it also to accept −3. Similarly, if it accepts 100 it should also accept 2,345 as a positive integer. Note that we are treating zero as a special case. We could, if we chose to, include zero with the positive integers, but my rudimentary specification did not specify that clearly, so it is really left as an undefined value (and it is not untypical to find such ambiguities or undefined areas in specifications). It often suits us to treat zero as a special case for testing where ranges of numbers are involved; we treat it as an equivalence partition with only one member. So we have three valid equivalence partitions in this case.

The equivalence partitioning technique takes advantage of the properties of equivalence partitions to reduce the number of test cases we need to write. Since all the values in an equivalence partition are handled in exactly the same way by a given program, we need only test one of them as a representative of the partition. In the example given, then, we need any positive integer, any negative integer and zero. We generally select values somewhere near the middle of each partition, so we might choose, say, −5,000, 0 and 5,000 as our representatives.

These three test inputs would exercise all three partitions and the theory tells us that if the program treats these three values correctly it is very likely to treat all of the other values, all 19,998 of them in this case, correctly. The partitions we have identified so far are called valid equivalence partitions because they partition the collection of valid

inputs, but there are other possible inputs to this program that would not be valid – real numbers, for example. We also have two input partitions of integers that are not valid: integers less than –10,000 and integers greater than 10,000. We should test that the program does not accept these, which is just as important as the program accepting valid inputs.

Non-valid partitions are also important to test. If you think about the example we have been using you will soon recognize that there are far more possible non-valid inputs than valid ones, since all the real numbers (e.g. numbers containing decimals) and all characters are non-valid in this case. It is generally the case that there are far more ways to provide incorrect input than there are to provide correct input; as a result, we need to ensure that we have tested the program against the possible non-valid inputs. Here again equivalence partitioning comes to our aid: all real numbers are equally non-valid, as are all alphabetic characters.

These represent two non-valid partitions that we should test, using values such as 9.45 and 'r' respectively. There will be many other possible non-valid input partitions, so we may have to limit the test cases to the ones that are most likely to crop up in a real situation.

Output partitions

Just as the input to a program can be partitioned, so can the output. The program in the exercise above could produce outputs of 0.5 per cent, 1 per cent and 1.5 percent. so we could use test cases that generate each of these outputs as an alternative to generating input partitions. An input value in the range £0.00–£1,000.00 would generate the 0.5 per cent output; a value in the range £1,001.00–£2,000.00 would generate the 1 per cent output; a value greater than £2,000.00 would generate the 1.5 per cent output.

Boundary Value Analysis

One thing we know about the kinds of mistakes that programmers make is that errors tend to cluster around boundaries. For example, if a program should accept a sequence

of numbers between 1 and 10, the most likely fault will be that values just outside this range are incorrectly accepted or that values just inside the range are incorrectly rejected. In the programming world, these faults coincide with particular programming structures such as the number of times a program loop is executed or the exact point at which a loop should stop executing.

Boundary Value Analysis (BVA) is a black box testing technique in which input values are selected near the boundaries. Boundary value analysis is employed based on the premise that errors occur more frequently for extreme values of a variable. Often, behaviour at the edge of a boundary is more likely to be incorrect, so boundaries are an area where testing is likely to yield defects.

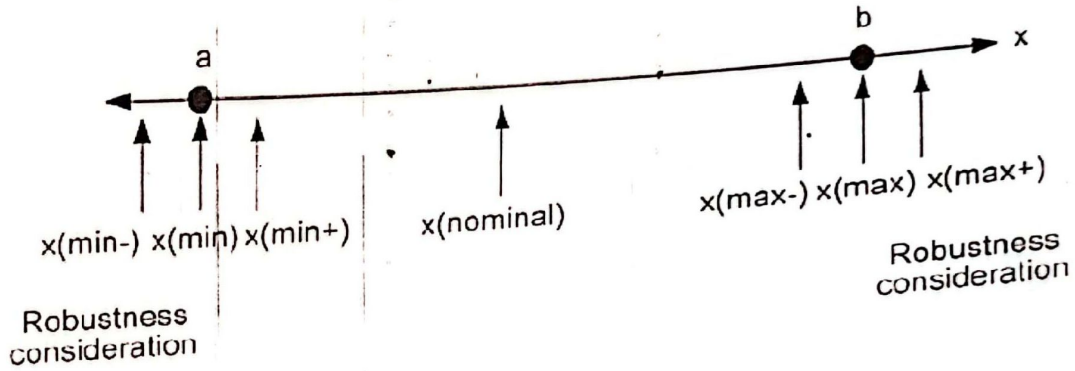
A partition of integers between 1 and 99, for instance, has a lowest value, 1, and a highest value, 99. These are called boundary values. Actually they are called valid boundary values because they are the boundaries on the inside of a valid partition. What about the values on the outside? Yes, they have boundaries too. So the boundary of the non-valid values at the lower end will be zero because, it is the first value you come to when you step outside the partition at the bottom end. You can also think of this as the highest value inside the non-valid partition of integers that are less than one. At the top end of the range we also have a non-valid boundary value, 100. This is the boundary value technique, more or less. For most practical purposes the boundary value analysis technique needs to identify just two values at each boundary. For this variant, we include one more value at each boundary when we use boundary value analysis:

> *the rule is that we use the boundary value itself and one value (as close as you can get) either side of the boundary. So, in this case lower boundary values will be 0, 1, 2 and upper boundary values will be 98, 99, 100. What does 'as close as we can get' mean? It means that, take the next value in sequence using the precision that has been applied to the partition. If the numbers are to a precision of 0.01, for*

example, the lower boundary values would be 0.99, 1.00, 1.01 and the upper boundary values would be 98.99, 99.00, 99.01.

In boundary value analysis, the critical input value is selected to generate the required test cases. Therefore, for a range of values bounded by 'a' and 'b', as a rule in BVA, the tester is expected to test (a-1), a, (a+1), (b-1), b, and (b+1).

The figure below further illustrates the selection of values that needs to be tested around the boundary of a given range of values.



Illustrative Example for Boundary Value Selection

A boundary value for a valid partition is a valid boundary value; the boundary of an invalid partition is an invalid boundary value. Tests can be designed to cover both valid and invalid boundary values to ensure the robustness of the software under test.

Boundary value analysis can be applied at all test levels. It is relatively easy to apply and its defect finding capability is high: Observing from a different perspective, boundary value analysis can be seen as an extension to equivalence partitioning, although the consideration are more focused at the boundaries.

The following examples shed more light on the idea of boundaries:

Example 1: The boiling point of water – the boundary is at 100 degrees Celsius, so for the 3 Value Boundary approach the boundary values will be 99 degrees, 100 degrees, 101

degrees – unless you have a very accurate digital thermometer, in which case they could be 99.9 degrees, 100.0 degrees, 100.1 degrees.

Example 2: Grading of scores – if an exam has a pass boundary at 40%, merit at 60% and distinction at 80% the 3 value boundaries would be 39, 40, 41 for pass, 59, 60, 61 for merit, 79, 80, 81 for distinction. It is unlikely that marks would be recorded at any greater precision than whole numbers.

It can be deduced from the illustrations given above that, BVA leads to selection of test cases that exercise boundary values; the technique complements equivalence partitioning. Rather than selecting any value or data in an equivalence class, the rule of the BVA is to select those at the "edge" of the class.

6. Software Maintenance

Why is maintenance needed for software, when software has no physical components that can degrade with age? Software needs to be maintained because of the residual defects remaining in the system. It is commonly believed that the state of the art today is limited and developing software with zero defect density is not possible. These defects, once discovered, need to be removed. Maintenance is also needed to change the delivered software to satisfy the enhanced needs of the users and the environment. Over the life of a software system, maintenance cost can far exceed the cost of original development

Types of Maintenance

Maintenance may be classified as:

(*i*) *Corrective Maintenance.* Corrective maintenance means repairing processing or performance failures or making changes because of previously uncorrected problems.

(*ii*) *Adaptive Maintenance.* Adaptive maintenance means changing the program function. This is done to adapt to the external environment change. For example, the current system was designed so that it calculates taxes on profits after deducting the dividend on equity shares. The government has issued orders now to include the dividend in the company profit for tax calculation. This function needs to be changed to adapt to the new system.

(*iii*) *Perfective Maintenance.* Perfective maintenance means enhancing the performance or modifying the programs to respond to the user's additional or changing needs. For example, earlier data was sent from stores to headquarters on magnetic media but after the stores were electronically linked via leased lines, the software was enhanced to send data via leased lines.

As maintenance is very costly and very essential, efforts have been done to reduce its costs. One way to reduce the costs is through maintenance management and software modification audits. Software modification consists of program rewriting and system-level-upgrading.

(*iv*) *Preventive Maintenance.* Preventive maintenance is the process by which we prevent our system from being obsolete. Preventive maintenance involves the concept of re-engineering and reverse engineering in which an old system with an old technology is re-engineered using new technology. This maintenance prevents the system from dying out.

THE END