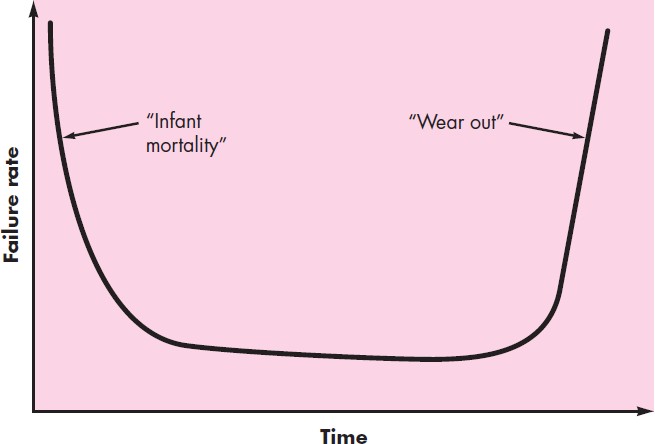
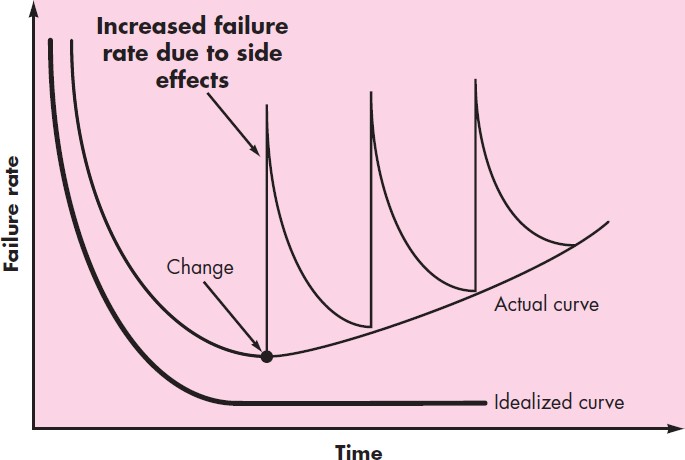
**UNIT –I INTRODUCTION TO SE**

1. **Software characteristics:**



* 1. *Software is developed or engineered; it is not manufactured in the classical sense.*

In both software development and hardware manufacturing activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent for software. Both activities require the construction of a “product,” but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

* 1. *Software doesn’t “wear out.”*

Figure 1, depicts failure rate as a function of time for hardware. The relationship, often called the “bathtub curve,” indicates that hardware exhibits relatively high failure rates early in its life due to manufacturing defects; defects are corrected and the failure rate drops to a steady-state level for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, temperature extremes, etc. Stated simply, the hardware begins to wear out.

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, the failure rate curve for software should take the form of the “idealized curve” shown in Figure 2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. Software doesn’t wear out. But it does deteriorate! During its life, software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the “actual curve”.

Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Fig:1 depicts failure rate as a function of time for hardware. Fig:2 failure rate curve for software.

* 1. *Although the industry is moving toward component-based construction, most software continues to be custom built.*

Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today’s interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

### Software Application Domains:

Today, seven broad categories of computer software present continuing challenges for software engineers:

*System software*—a collection of programs written to service other programs. Some system software (e.g., compilers, and editors) processes complex, but determinate, information structures. Other systems applications (e.g., operating system components, networking software) process largely indeterminate data. In either case, the systems software area is characterized by heavy interaction with computer hardware;

*Application software*—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. Application software is used to control business functions in real time (e.g., point-of-sale transaction processing, real-time manufacturing process control).

*Engineering/scientific software*—has been characterized by “number crunching” algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.

*Embedded software*—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven).

*Product-line software*—designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management).

*Web applications*—called “WebApps,” in their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

*Artificial intelligence software*—makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

1. **Software myths-** erroneous beliefs about software and the process that is used to build it. They appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who “know the score.” Today, most knowledgeable software engineering professionals recognize myths for what they are- misleading attitudes that have caused serious problems for managers and practitioners alike. **Management myths-** Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.

*Myth:* We already have a book that’s full of standards and procedures for building software. Won’t that provide my people with everything they need to know?

*Reality:* The book of standards may very well exist, but is it used? Are software

practitioners aware of its existence? Does it reflect modern software engineering practice? In many cases, the answer to all of these questions is “no.”

*Myth:* If we get behind schedule, we can add more programmers and catch up

*Reality:* As new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well coordinated manner.

*Myth:* If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

*Reality:* If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

**Customer myths-** the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations, and dissatisfaction with the developer.

*Myth:* A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.

*Reality:* a comprehensive and stable statement of requirements is not always possible.

Unambiguous requirements are developed only through effective and continuous communication. *Myth:* Software requirements continually change, but change can be easily accommodated because software is flexible.

*Reality:* the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small. However, as time passes, the cost impact grows rapidly.

**Practitioner’s myths-** myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. Old ways and attitudes die hard.

*Myth:* Once we write the program and get it to work, our job is done.

*Reality:* Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

*Myth:* Until I get the program “running” I have no way of assessing its quality.

*Reality:* One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the technical review.

*Myth:* The only deliverable work product for a successful project is the working program. *Reality:* A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

*Myth:* Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

*Reality:* Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

### Some Short Notes:

***Legacy software:*** older programs—often referred to as legacy software. This is causing headaches for large organizations who find them costly to maintain and risky to evolve. There is sometimes one additional characteristic that is present in legacy software—poor quality. Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, a poorly managed change history.

***Role of software*:** software has dual role: product and as the vehicle for delivering a product.

***Software:*** is: (1) instructions that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

***Process flow:*** describes how the framework activities ( communication, planning, modeling, construction, deployment) and the actions and tasks that occur within each framework activity are organized with respect to sequence and time.

### Process Framework:

A **software process** is a framework for the activities, actions, and tasks that are required to build high-quality software. Each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a task set that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

A **generic process framework** for software engineering defines five framework activities— communication, planning, modeling, construction, and deployment. In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

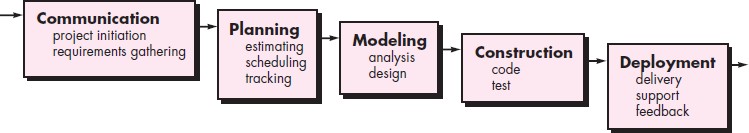
**Process flow**—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time. A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment . An iterative process flow repeats one or more of the activities before

proceeding to the next. An evolutionary process flow executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software. A parallel process flow executes one or more activities in parallel with other activities.

### PROCESS MODELS

### a)The Waterfall Model:

It can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner. There are times when the requirements for a problem are well understood—when work flows from communication through deployment in a reasonably linear fashion.

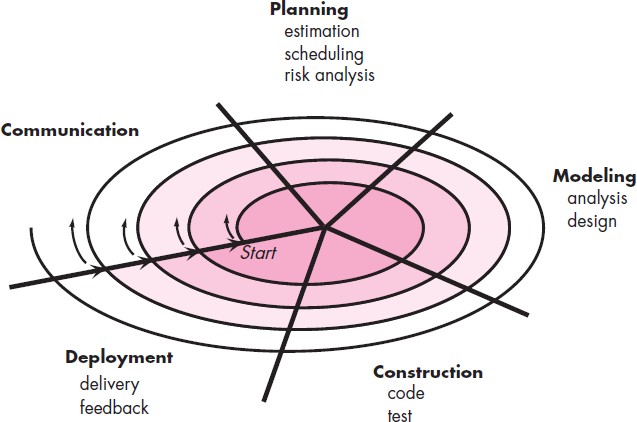


This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made. The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software. Among the problems that are sometimes encountered when the waterfall model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes.
2. It is often difficult for the customer to state all requirements explicitly
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span.

### b) The Spiral Model:

The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software.



Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, enables you to apply the prototyping approach at any stage in the evolution of the product.

# c) AGILE DEVELOPMENT

## WHAT IS AGILITY?

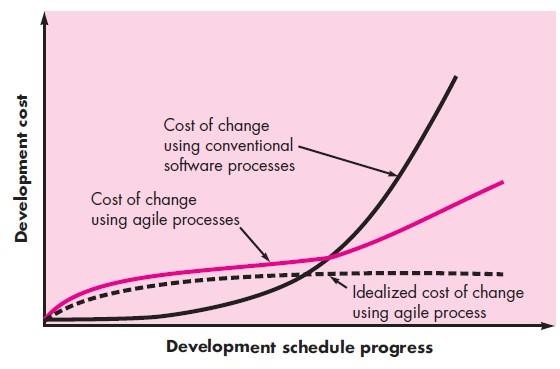
**Agile** is a software development methodology to build software incrementally using short iterations of 1 to 4 weeks so that the development process is aligned with the changing business needs.

An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.

## AGILITY AND THE COST OF CHANGE

An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment.

Agility argue that a well-designed agile process “flattens” the cost of change curve shown in following figure, allowing a software team to accommodate changes late in a software project without dramatic cost and time impact. When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming, the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence to suggest that a significant reduction in the cost of change can be achieved.



## AGILE PROCESS

Any agile software process is characterized in a manner that addresses a number of key assumptions about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable

##### Agility Principles

Agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self– organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then

tunes and adjusts its behavior accordingly.

High smith argues that an agile, adaptive development approach based on collaboration is “as much a source of *order* in our complex interactions as discipline and engineering.” He defines an ASD “life cycle” that incorporates three phases, speculation, collaboration, and learning.

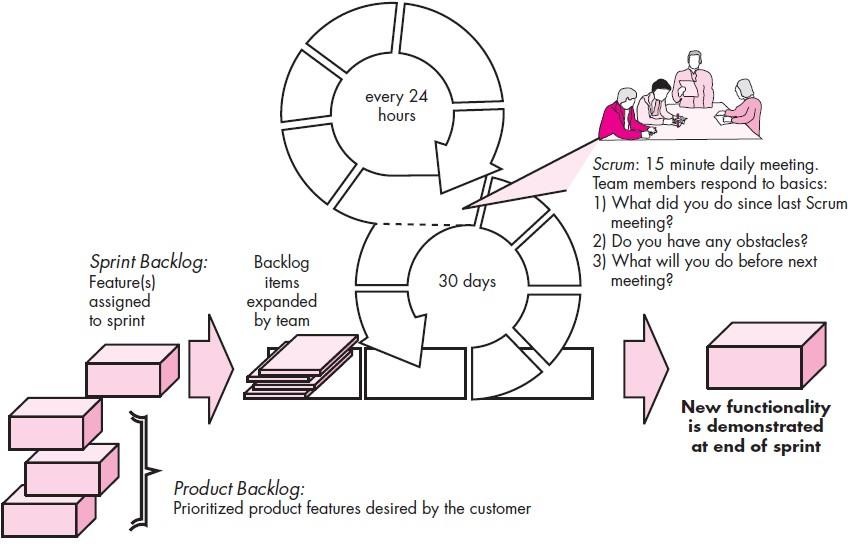
As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on **“learning**” as much as it is on progress toward a completed cycle.

ASD teams learn in **three** ways: **focus groups, technical reviews , and project postmortems**. ASD’s overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

##### Scrum

Scrum is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern called a ***sprint.*** The work conducted within a sprint is adapted to the problem at hand and is defined and often modified

in real time by the Scrum team. The overall flow of the Scrum process is illustrated in following figure



**Scrum** emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development actions:

* ***Backlog***—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time. The product manager assesses the backlog and updates priorities as required.
* ***Sprints***—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box (typically 30 days). Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.
* ***Scrum meetings***—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members
  + What did you do since the last team meeting?
  + What obstacles are you encountering?
  + What do you plan to accomplish by the next team meeting?

A team leader, called a ***Scrum master****,* leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to “**knowledge socialization**”

* ***Demos***—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

### UNIT-II Requirements, Requirements Engineering

1. **Functional and non-functional requirements:**

Software requirements are often classified as functional and non-functional. *Functional requirements* are statements of services the systems should provide, how the system should react particular inputs, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, these may also explicitly state what the system should not do. These depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements functional requirements usually described in an abstract way that can be understood by system users. The functional user requirements define specific facilities to be provided by the system. Functional system requirements vary from general requirements covering what the system should do to very specific requirements.

***Non-functional requirements:*** These are constraints on the services (or) functions offered by the system. They include timing constraints, constraints on development process, and constraints imposed by standards. These often apply to the system as a whole, rather than individual system features (or) services. These may affect the overall architecture of a system rather than individual components.

***Types of non-functional requirements-***

*Product requirements*- specify or constrain the behavior of the software. Examples: performance- how fast it must execute, how much memory it needs, etc.

*Organisational requirements*- are derived from policies and procedures in the customer's and developer's oranization. Example: how system will be used (operational process requirements) *External requirements*- are derived from the factors external to the system and development process. Examples: interoperability requirements, legislative requirements, etc.

***Requirements engineering***- the process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed. The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process. It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.

### User requirements/ System requirements:

*User requirements*- the high-level abstract statements in a natural language and using diagrams of what services the system is expected to provide to system users and constrainsts under which it must operate. User requirements are quite general.

*System requirements*- are more detailed descriptions of the software system's functions, services and operational constrainsts. The system requirements documents should define exactly what is to be implemented. The system requirements provvide more specific information about the services and functions of the system that is to be implemented.

Example: Mental health care patient management system (MHC-PMS) User Requirements Definition

* 1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

* 1. On the last working day of each month, a summary of drugs prescribed, their cost, and prescribing clincs shall be generated.
  2. The system shall automatically generate the report for printing after 17:30 on the last working day of the month.
  3. A report shall be created for each clinic and shall list drug names, the total number of prescriptions, the number of doeses and the total cost of the drugs.
  4. If drugs are available in different dose units, separate reprots shall be created for each dose unit.
  5. Access to all cost reports shall be restricted to authorized users listed on a management access control list.

### Requirements engineering process:

System requirements specification and

modeling

Requirements specification

User requirements specification

Business requirements specification

System requirements elicitation

User requirements elicitation

Feasibility study

Prototyping

Requirements elicitation

Reviews

Requirements validation

**Sysetm requirements document**

Fig: shows the interleaving of activities to carry out requirements engineering process.

This includes four high-level activities: feasibility *study, elicitation and analysis, specification, and validation*. Feasibility study consists of assessing if the system is useful to the business or not. The purpose of requirements elicitation and analysis is for discovering requirements. Specification means converting the requirements into some standard form. Validation involves checking that the requirements actually define the system that the customer wants.

In practice, requirements engineering is an iterative process in which the activities are interleaved.

In the above diagram, activities are organized as an iterative process around a spiral, with output being a system requirements document. Early in the process, most effort will be spent on understanding high-level business and non-functional requirements, and the user requirements for the system. Later in the process, in the outer rings of the spiral, most effort will be devoted to eliciting and understanding the detailed system requirements. The number of iterations around the spiral can vary so the spiral can be exited after some or all of the user requirements have been elicited.

### Requirements elicitation and analysis:

After an initial feasibility study, the next stage of requirements engineering process is requirements elicitation and analysis. Here, software engineers work with all stakeholders such as customers and system end-users to find out the application domain, what services the system should provide, the required performance, hardware constraints and so on. A system stakeholder is anyone who should have some direct or indirect influence on the system.

Elicitation and analysis consists of different activities: Requirements discovery, Requirements classification and organization, prioritization and negotiation, and specification.

*Discovery*- it is the process of interacting with stakeholders of the system to discover requirements. Domain requirements from stakeholders and documentations are also discovered during this activity.

*Classification and organization:* It takes unstructured collection of requirements, groups related requirements and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system.

*Prioritization and negotiation:* when multiple stakeholders are involved, requirements will conflict. It is concerned with prioritizing requirements, and finding and resolving requirements conflicts through negotiation. Usually stakeholders have to meet to resolve differences and agree an compromise requirements.

*Specification:* requirements are documented and input into the next round of the spiral. Formal and informal documents may be produced.

### Validation techniques:

*Reviews:* the requirements are analyzed systematically by a team of reviews who check for errors and inconsistencies.

*Prototyping:* an executable model of the system that is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.

*Test case generation:* requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult, this means requirements will be difficult to implement.

### UNIT-III

# Design Concepts

**Introduction:** Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. Design principles establish an overriding philosophy that guides you in the design work you must perform. Design is pivotal to successful software engineering

The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight Software design changes continually as new methods, better analysis, and broader understanding evolve

### DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

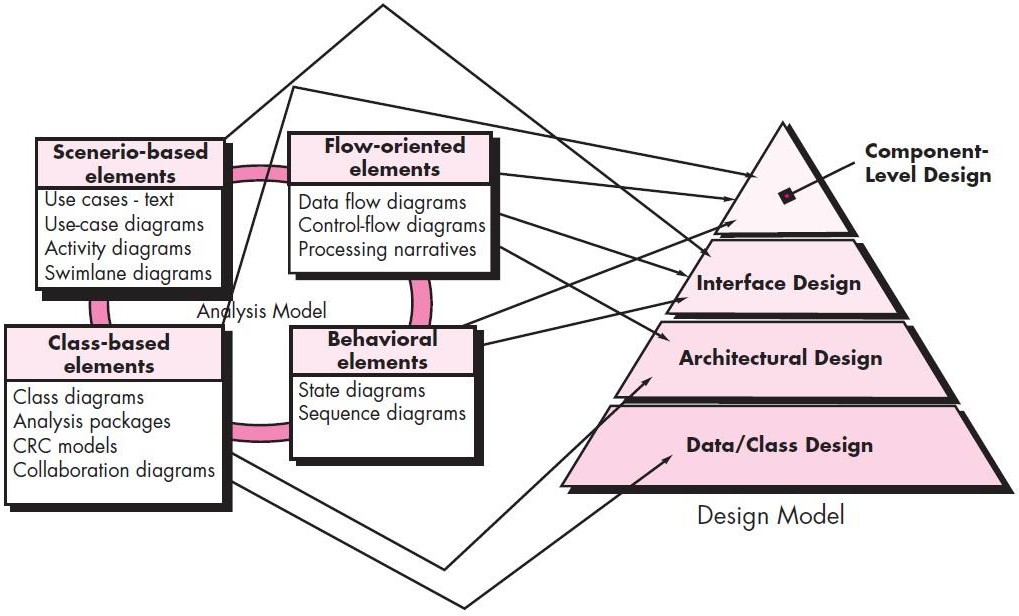
Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).

Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in following figure.

The requirements model, manifested by **scenario-based, class-based, flow-oriented, and behavioral elements,** feed the design task.

The **data/class design** transforms class models into design class realizations and the requisite data structures required to implement the software.

The **architectural design** defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer- based system—is derived from the requirements model.



##### Fig : Translating the requirements model into the design model

The **interface design** describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The **component-level design** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

The importance of software design can be stated with a single word—***quality*.** Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder’s requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow.

### THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a “**blueprint**” for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a **high level of abstraction**

##### Software Quality Guidelines and Attributes

McGlaughlin suggests **three** characteristics that serve as a guide for the evaluation of a good design:

* The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
* The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
* The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

**Quality Guidelines.** In order to evaluate the quality of a design representation, consider the following guidelines:

* 1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion,2 thereby facilitating implementation and testing.
  2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
  3. A design should contain distinct representations of data, architecture, interfaces, and components.
  4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
  5. A design should lead to components that exhibit independent functional characteristics.
  6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
  7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
  8. A design should be represented using a notation that effectively communicates its meaning.

**Quality Attributes.** Hewlett-Packard developed a set of software quality attributes that has been given the acronym **FURPS—functionality, usability, reliability, performance, and supportability**. The **FURPS** quality attributes represent a target for all software design:

* ***Functionality*** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system..
* ***Usability*** is assessed by considering human factors, overall aesthetics, consistency, and documentation.
* ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
* ***Performance*** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
* ***Supportability*** combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, ***maintainability***— and in addition, testability, compatibility, configurability, the ease with which a system can be installed, and the ease with which problems can be localized.

##### The Evolution of Software Design

The evolution of software design is a continuing process that has now spanned almost six decades. Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top down manner. Procedural aspects of design definition evolved into a philosophy called ***structured programming****.*

A number of design methods, growing out of the work just noted, are being applied throughout the industry. All of these methods have a number of common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

### DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the history of software engineering. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you answer the following questions:

* What criteria can be used to partition software into individual components?
* How is function or data structure detail separated from a conceptual representation of the software?
* What uniform criteria define the technical quality of a software design?

The following brief overview of important software design concepts that span both traditional and object-oriented software development.

##### Abstraction

*Abstraction* is the act of representing essential features without including the background details or explanations. the *abstraction* is used to reduce complexity and allow efficient design and implementation of complex *software* systems. Many levels of abstraction can be posed. At the **highest level** of abstraction, a solution is stated in broad terms using the language of the problem environment. At **lower levels** of abstraction, a more detailed description of the solution is provided.

As different levels of abstraction are developed, you work to create both **procedural** and

##### data abstractions.

A ***procedural abstraction*** refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed.

A ***data abstraction*** is a named collection of data that describes a data object.

##### Architecture

***Software architecture*** alludes to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system” Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

Shaw and Garlan describe a set of properties that should be specified as part of an architectural design:

* **Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another.
* **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
* **Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

The architectural design can be represented using one or more of a number of different models. ***Structural models:*** *R*epresent architecture as an organized collection of program components. ***Framework models:*** Increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

***Dynamic models*** *:* Address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

***Process models*** *:*Focus on the design of the business or technical process that the system must accommodate.

***Functional models*** can be used to represent the functional hierarchy of a system.

A number of different ***architectural description languages* (ADLs)** have been developed to represent these models.

##### Patterns

Brad Appleton defines a ***design pattern*** in the following manner: “A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns”

A design pattern describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

##### Separation of Concerns

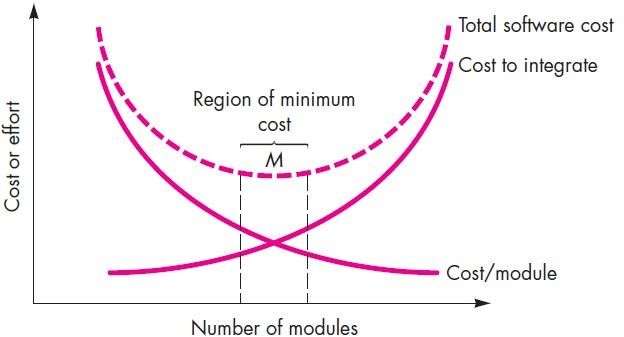
***Separation of concerns*** is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is a feature or behavior that is specified as part of the requirements model for the software.

Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow.

##### Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called ***module.***

Modularity is the single attribute of software that allows a program to be intellectually manageable



##### Fig : Modularity and software cost

**Information Hiding**

The principle of information hiding suggests that modules be “characterized by design decisions that hides from all others.” In other words, modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

##### Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with “**single minded**” function and an “aversion” to excessive interaction with other modules.

Independence is assessed using **two** qualitative criteria: **cohesion** and **coupling**. ***Cohesion*** is an indication of the relative functional strength of a module. ***Coupling*** is an indication of the relative interdependence among modules.

**Cohesion** is a natural extension of the information-hiding concept. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should do just one thing. Although you should always strive for **high cohesion** (i.e., single-mindedness).

**Coupling** is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the **lowest possible coupling**.

##### Refinement

Stepwise refinement is a **top-down** design strategy originally proposed by Niklaus Wirth. Refinement is actually a process of ***elaboration****.* You begin with a statement of function that is defined at a high level of abstraction.

**Abstraction** and **refinement** are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for “outsiders” to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses.

##### Aspects

An ***aspect*** is a representation of a crosscutting concern. A crosscutting concern is some characteristic of the system that applies across many different requirements.

##### Refactoring

An important design activity suggested for many agile methods, ***refactoring*** is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. **Fowler** defines refactoring in the following manner: “**Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”**

##### Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering. OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.

##### Design Classes

The requirements model defines a set of analysis classes. Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. A set of ***design classes*** that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.

**Five** different types of design classes, each representing a different layer of the design architecture, can be developed:

* ***User interface classes*** define all abstractions that are necessary for human computer interaction (HCI). The design classes for the interface may be visual representations of the elements of the metaphor.
* ***Business domain classes*** are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
* ***Process classes*** implement lower-level business abstractions required to fully manage the business domain classes.
* ***Persistent classes*** represent data stores (e.g., a database) that will persist beyond the execution of the software.
* ***System classes*** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

Arlow and Neustadt suggest that each design class be reviewed to ensure that it is “**well- formed**.” They define **four** characteristics of a well-formed design class:

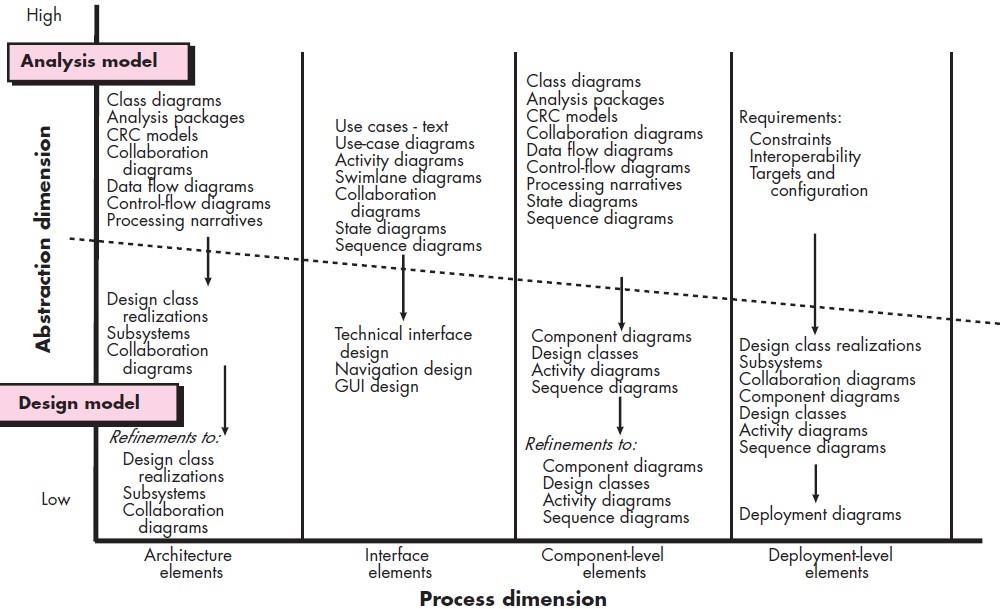
* + - * **Complete and sufficient.** A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.
      * **Primitiveness.** Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.
      * **High cohesion.** A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.
      * **Low coupling.** Within the design model, it is necessary for design classes to collaborate with one another. If a design model is highly coupled, the system is difficult to implement, to test, and to maintain over time.

### THE DESIGN MODEL

The design model can be viewed in **two** different dimensions. The ***process dimension*** indicates the evolution of the design model as design tasks are executed as part of the software process. The ***abstraction dimension*** represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. The design model has **four** major elements: data, architecture, components, and interface.

##### 3.4.1. Data Design Elements

Data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user’s view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. The structure of data has always been an important part of software design. At the program **component level**, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high- quality applications. At the **application level**, the translation of a data model into a database is pivotal to achieving the business objectives of a system. At the **business level**, the collection of

information stored in disparate databases and reorganized into a “data warehouse” enables data mining or knowledge discovery that can have an impact on the success of the business itself.

##### Fig : Dimensions of the design model