Introduction to Software Engineering

Software is a program or set of programs containing instructions that provide desired functionality. And Engineering is the process of designing and building something that serves a particular purpose and finds a cost-effective solution to problems.
Software Engineering is a systematic, disciplined, quantifiable study and approach to the design, development, operation, and maintenance of a software system.
The IEEE fully defines software engineering as: ... The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software

Why is Software Engineering required?
Software Engineering is required due to the following reasons:
To manage Large software
For more Scalability
Cost Management
To manage the dynamic nature of software
For better quality Management

Characteristic of software / Need of Software Engineering

The necessity of software engineering appears because of a higher rate of progress in user requirements and the environment on which the program is working.

•Efficiency –
The software should not make wasteful use of computing devices such as memory, processor cycles, etc.
•Correctness –
A software product is correct if the different requirements as specified in the SRS document have been correctly implemented.
•Reusability –
A software product has good reusability if the different modules of the product can easily be reused to develop new products.
•Testability –
Here software facilitates both the establishment of test criteria and the evaluation of the software with respect to those criteria.
•Maintainability –
It should be feasible for the software to evolve to meet changing requirements.
•Reliability –
It is an attribute of software quality. The extent to which a program can be expected to perform its desired function, over an arbitrary time period.
•Portability –

In this case, the software can be transferred from one computer system or environment to another.
•Adaptability –
In this case, the software allows different system constraints and the user needs to be satisfied by making changes to the software.
•Interoperability – Capability of 2 or more functional units to process data cooperatively.

The necessity of Software evolution: Software evaluation is necessary just because of the following reasons:
a) Change in requirement with time: With the passes of time, the organization's needs and modus Operandi of working could substantially be changed so in this frequently changing time the tools(software) that they are using need to change for maximizing the performance.
b) Environment change: As the working environment changes the things(tools) that enable us to work in that environment also changes proportionally same happens in the software world as the working environment changes then, the organizations need reintroduction of old software with updated features and functionality to adapt the new environment.
c) Errors and bugs: As the age of the deployed software within an organization increases their preciseness or impeccability decrease and the efficiency to bear the increasing complexity workload also continually degrades. So, in that case, it becomes necessary to avoid use of obsolete and aged software. All such obsolete Softwares need to undergo the evolution process in order to become robust as per the workload complexity of the current environment.
d) Security risks: Using outdated software within an organization may lead you to at the verge of various software-based cyberattacks and could expose your confidential data illegally associated with the software that is in use. So, it becomes necessary to avoid such security breaches through regular assessment of the security patches/modules are used within the software. If the software isn't robust enough to bear the current occurring Cyber attacks so it must be changed (updated).
e) For having new functionality and features: In order to increase the performance and fast data processing and other functionalities, an organization need to continuously evolute the software throughout its life cycle so that stakeholders & clients of the product could work efficiently.

Changing Nature of Software:
Nowadays, seven broad categories of computer software present continuing challenges for software engineers .which is given below:
System Software:
System software is a collection of programs which are written to service other programs. Some system software processes complex but determinate, information structures. Other system application process largely indeterminate data. Sometimes when, the system software area is characterized by the heavy interaction with

computer hardware that requires scheduling, resource sharing, and sophisticated process management.

Application Software:

Application software **i**s defined as programs that solve a specific business need. Application in this area process business or technical data in a way that facilitates business operation or management technical decision making. In addition to convention data processing application, application software is used to control business function in real time.

Engineering and Scientific Software:

This software is used to facilitate the engineering function and task. however modern application within the engineering and scientific area are moving away from the conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take a real-time and even system software characteristic.

Embedded Software:

Embedded software resides within the system or product and is used to implement and control feature and function for the end-user and for the system itself. Embedded software can perform the limited and esoteric function or provided significant function and control capability.

Product-line Software:

Designed to provide a specific capability for use by many different customers, product line software can focus on the limited and esoteric marketplace or address the mass consumer market.

Web Application:

It is a client-server computer program which the client runs on the web browser. In their simplest form, Web apps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, as e-commerce and B2B application grow in importance. Web apps are evolving into a sophisticate computing environment that not only provides a standalone feature, computing function, and content to the end user.

Artificial Intelligence Software:

Artificial intelligence software makes use of a nonnumerical algorithm to solve a complex problem that is not amenable to computation or straightforward analysis. Application within this area includes robotics, expert system, pattern recognition, artificial neural network, theorem proving and game playing.

Legacy Software

Legacy Software Systems were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation (rapid increase in the number or amount of something) of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

Characteristics of Legacy Software

• Longevity (Lifetime) : Very high • Business criticality : High • Poor quality • Convoluted code • Poor or nonexistent documents, test cases.

Changes needed in legacy software

1. The software must be adopted to meet the needs of new computing environments or technology. 2. The software must be enhanced to implement new business requirements. 3. The software must be extended to make it interoperable with more modern systems or databases. 4. The software must be re-architected to make it viable within a network environment.

Software Myths :

(i) Management Myths :

Myth 1:

We have all the standards and procedures available for software development i.e. the software developer has all the reqd.

Fact :

Software experts do not know that there are all of them levels.Such practices may or may not be expired at present / modern software engineering methods.And all existing processes are incomplete.

Myth 2 :

The addition of the latest hardware programs will improve the software development.

Fact:

The role of the latest hardware is not very high on standard software development; instead (CASE) Engineering tools help the computer they are more important than hardware to produce quality and productivity.

Hence, the hardware resources are misused.

Myth 3 :

Managers think that, with the addition of more people and program planners to Software development can help meet project deadlines (If lagging behind).

Fact :

Software development is not, the process of doing things like production; here the addition of people in previous stages can reduce the time it will be used for productive development, as the newcomers would take time existing developers of definitions and understanding of the file project. However, planned additions are organized and organized It can help complete the project.

(ii)Customer Myths :

The customer can be the direct users of the software, the technical team, marketing / sales department, or other company. Customer has myths

Leading to false expectations (customer) & that’s why you create dissatisfaction with the developer.

Myth 1 :

A general statement of intent is enough to start writing plans (software development) and details of objectives can be done over time.
Fact:
Official and detailed description of the database function, ethical performance, communication, structural issues and the verification process are important.
It is happening that the complete communication between the customer and the developer is required.
(iii)Practitioner's Myths :
Myths 1 :
They believe that their work has been completed with the writing of the plan and they received it to work.
Fact:
It is true that every 60-80% effort goes into the maintenance phase (as of the latter software release). Efforts are required, where the product is available first delivered to customers.
Myths 2 :
There is no other way to achieve system quality, behind it done running.
Fact:
Systematic review of project technology is the quality of effective software verification method. These updates are quality filters and more accessible than test.

Three Phases of the Generic View of Software Engineering:

01. Definition Phase:

The definition phase focuses on "what". That is, during definition, the software engineer attempts to identify what information is to be processed, what function and performance are desired, what system behavior can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system. During this, three major tasks will occur in some form: system or information engineering, software project planning and requirements analysis.

02. Development Phase:

The development phase focuses on "how". That is, during development a software engineer attempts to define how data are to be structured, how function is to be implemented within a software architecture, how interfaces are to be characterized, how the design will be translated into a programming language, and how testing will be performed. During this, three specific technical tasks should always occur; software design, code generation, and software testing.

03. Support Phase:

The support phase focuses on “change” associated with error correction, adaptations required as the software’s environment evolves, and changes due to enhancements brought about by changing customer requirements. Four types of change are encountered during the support phase:

Program vs Software Product:

•A program is a set of instructions that are given to a computer in order to achieve a specific task whereas software is when a program is made available for commercial business and is properly documented along with its licensing. Software= Program + documentation + licensing.

•A program is one of the stages involved in the development of the software, whereas a software development usually follows a life cycle, which involves the feasibility study of the project, requirement gathering, development of a prototype, system design, coding, and testing.

Software Development Life Cycle (SDLC)

A software life cycle model (also termed process model) is a pictorial and diagrammatic representation of the software life cycle. A life cycle model represents all the methods required to make a software product transit through its life cycle stages. It also captures the structure in which these methods are to be undertaken.

Need of SDLC

The development team must determine a suitable life cycle model for a particular plan and then observe to it.

Without using an exact life cycle model, the development of a software product would not be in a systematic and disciplined manner. When a team is developing a software product, there must be a clear understanding among team representative about when and what to do. Otherwise, it would point to chaos and project failure. This problem can be defined by using an example. Suppose a software development issue is divided into various parts and the parts are assigned to the team members. From then on, suppose the team representative is allowed the freedom to develop the roles assigned to them in whatever way they like. It is possible that one representative might start writing the code for his part, another might choose to prepare the test documents first, and some other engineer might begin with the

design phase of the roles assigned to him. This would be one of the perfect methods for project failure.

SDLC Cycle

SDLC Cycle represents the process of developing software. SDLC framework includes the following steps:

Stage1: Planning and requirement analysis

Requirement analysis is significant and essential activity after elicitation. We analyze, refine, and scrutinize the gathered requirements to make consistent and unambiguous requirements. This activity reviews all requirements and may provide a graphical view of the entire system. After the completion of the analysis, it is expected that the understandability of the project may improve significantly. Here, we may also use the interaction with the customer to clarify points of confusion and to understand which requirements are more important than others.**For Example**, A client wants to have an application which concerns money transactions. In this method, the requirement has to be precise like what kind of operations will be done, how it will be done, in which currency it will be done, etc.

Stage2: Defining Requirements

Once the requirement analysis is done, the next stage is to certainly represent and document the software requirements and get them accepted from the project stakeholders.

This is accomplished through "SRS"- Software Requirement Specification document which contains all the product requirements to be constructed and developed during the project life cycle.

Stage3: Designing the Software

The next phase is about to bring down all the knowledge of requirements, analysis, and design of the software project. This phase is the product of the last two, like inputs from the customer and requirement gathering.In this phase of SDLC, the actual development begins, and the programming is built. The implementation of design begins concerning writing code. Developers have to follow the coding guidelines described by their management and programming tools like compilers, interpreters, debuggers, etc. are used to develop and implement the code.

Stage5: Testing

After the code is generated, it is tested against the requirements to make sure that the products are solving the needs addressed and gathered during the requirements stage.During this stage, unit testing, integration testing, system testing, acceptance testing are done.

Stage6: Deployment

Once the software is certified, and no bugs or errors are stated, then it is deployed.Then based on the assessment, the software may be released as it is or with suggested enhancement in the object segment.After the software is deployed, then its maintenance begins.

Stage7: Maintenance

Once when the client starts using the developed systems, then the real issues come up and requirements to be solved from time to time.This procedure where the care is taken for the developed product is known as maintenance.

Capability Maturity Model Integration (CMMI)

Capability Maturity Model Integration (CMMI) is a successor of CMM and is a more evolved model that incorporates best components of individual disciplines of CMM like Software CMM, Systems Engineering CMM, People CMM, etc. Since CMM is a reference model of matured practices in a specific discipline, so it becomes difficult to integrate these disciplines as per the requirements. This is why CMMI is used as it allows the integration of multiple disciplines as and when needed.

Objectives of CMMI :

Fulfilling customer needs and expectations.
Value creation for investors/stockholders.
Market growth is increased.
Improved quality of products and services.
Enhanced reputation in Industry.

CMMI Representation – Staged and Continuous :

A representation allows an organization to pursue a different set of improvement objectives. There are two representations for CMMI :

Staged Representation :

uses a pre-defined set of process areas to define improvement path.provides a sequence of improvements, where each part in the sequence serves as a foundation for the next.an improved path is defined by maturity level.maturity level describes the maturity of processes in organization.Staged CMMI representation allows comparison between different organizations for multiple maturity levels.

Continuous Representation :

allows selection of specific process areas.uses capability levels that measures improvement of an individual process area.Continuous CMMI representation allows comparison between different organizations on a process-area-by-process-area basis.allows organizations to select processes which require more improvement.

Software process model

A software process model is an abstraction of the software development process. The models specify the stages and order of a process. So, think of this as a representation of the order of activities of the process and the sequence in which they are performed.

A model will define the following:

- The tasks to be performed
- The input and output of each task
- The pre and post conditions for each task
- The flow and sequence of each task

The goal of a software process model is to provide guidance for controlling and coordinating the tasks to achieve the end product and objectives as effectively as possible.

There are many kinds of process models for meeting different requirements. We refer to these as SDLC models (Software Development Life Cycle models). The most popular and important SDLC models are as follows:

Waterfall model, V model, Incremental model, RAD model, Agile model, Iterative model, Prototype model, Spiral model

Waterfall Model

The waterfall model is a sequential, plan driven-process where you must plan and schedule all your activities before starting the project. Each activity in the waterfall model is represented as a separate phase arranged in linear order.Classical waterfall model is the basic software development life cycle model. It is very simple but idealistic. Earlier this model was very popular but nowadays it is not used. But it is very important because all the other software development life cycle models are based on the classical waterfall model.

Classical waterfall model divides the life cycle into a set of phases. This model considers that one phase can be started after completion of the previous phase. That is the output of one phase will be the input to the next phase. Thus the development process can be considered as a sequential flow in the waterfall. Here the phases do not overlap with each other.

It has the following phases:

Requirements, Design, Implementation, Testing, Deployment, Maintenance

Each of these phases produces one or more documents that need to be approved before the next phase begins. However, in practice, these phases are very likely to overlap and may feed information to one another.

The waterfall model is easy to understand and follow. It doesn't require a lot of customer involvement after the specification is done. Since it's inflexible, it can't adapt to changes. There is no way to see or try the software until the last phase. The waterfall model has a rigid structure, so it should be used in cases where the requirements are understood completely and unlikely to radically change.

Advantage Of Waterfall model

•Classical waterfall model is an idealistic model for software development. It is very simple, so it can be considered as the basis for other software development life cycle models. Below are some of the major advantages of this SDLC model:
•This model is very simple and is easy to understand.
•Phases in this model are processed one at a time.
•Each stage in the model is clearly defined.
•This model has very clear and well understood milestones.
•Process, actions and results are very well documented.
•Reinforces good habits: define-before- design,
design-before-code.
•This model works well for smaller projects and projects where requirements are well understood.

Disadvantage of waterfall model

•Classical waterfall model suffers from various shortcomings, basically we can't use it in real projects, but we use other software development lifecycle models which are based on the classical waterfall model. Below are some major drawbacks of this model:
•No feedback path: In classical waterfall model evolution of software from one phase to another phase is like a waterfall. It assumes that no error is ever committed by developers during any phases. Therefore, it does not incorporate any mechanism for error correction.
•Difficult to accommodate change requests: This model assumes that all the customer requirements can be completely and correctly defined at the beginning of the project, but actually customers' requirements keep on changing with time. It is difficult to accommodate any change requests after the requirements specification phase is complete.
•No overlapping of phases: This model recommends that new phase can start only after the completion of the previous phase. But in real projects, this can't be maintained. To increase the efficiency and reduce the cost, phases may overlap.

Incremental Model

The incremental model divides the system's functionality into small increments that are delivered one after the other in quick succession. The most important functionality is implemented in the initial increments.
The subsequent increments expand on the previous ones until everything has been updated and implemented.
Incremental development is based on developing an initial implementation, exposing it to user feedback, and evolving it through new versions. The process' activities are interwoven by feedback.

The incremental model lets stakeholders and developers see results with the first increment. If the stakeholders don't like anything, everyone finds out a lot sooner. It is efficient as the developers only focus on what is important and bugs are fixed as they arise, but you need a clear and complete definition of the whole system before you start.
The incremental model is great for projects that have loosely-coupled parts and projects with complete and clear requirements.

Advantages of Incremental model:
Generates working software quickly and early during the software life cycle.
This model is more flexible – less costly to change scope and requirements.
It is easier to test and debug during a smaller iteration.
In this model customer can respond to each built.
Lowers initial delivery cost.
Easier to manage risk because risky pieces are identified and handled during it'd iteration.

Disadvantage of Incremental model
Needs good planning and design.
Needs a clear and complete definition of the whole system before it can be broken down and built incrementally.
Total cost is higher than waterfall.

Evolutionary model
Evolutionary model is a combination of Iterative and Incremental model of software development life cycle. Delivering your system in a big bang release, delivering it in incremental process over time is the action done in this model. Some initial requirements and architecture envisioning need to be done.
It is better for software products that have their feature sets redefined during development because of user feedback and other factors. The Evolutionary

development model divides the development cycle into smaller, incremental waterfall models in which users are able to get access to the product at the end of each cycle. Feedback is provided by the users on the product for the planning stage of the next cycle and the development team responds, often by changing the product, plan or process. Therefore, the software product evolves with time.
All the models have the disadvantage that the duration of time from start of the project to the delivery time of a solution is very high. Evolutionary model solves this problem in a different approach.

Evolutionary model suggests breaking down of work into smaller chunks, prioritizing them and then delivering those chunks to the customer one by one. The number of chunks is huge and is the number of deliveries made to the customer. The main advantage is that the customer's confidence increases as he constantly gets quantifiable goods or services from the beginning of the project to verify and validate his requirements. The model allows for changing requirements as well as all work in broken down into maintainable work chunks.
Application of Evolutionary Model:
It is used in large projects where you can easily find modules for incremental implementation. Evolutionary model is commonly used when the customer wants to start using the core features instead of waiting for the full software.
Evolutionary model is also used in object oriented software development because the system can be easily portioned into units in terms of objects.
Advantages:
In evolutionary model, a user gets a chance to experiment partially developed system.
It reduces the error because the core modules get tested thoroughly.
Disadvantages:
Sometimes it is hard to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented and delivered.

Iterative Model

The iterative development model develops a system through building small portions of all the features. This helps to meet initial scope quickly and release it for feedback. In the iterative model, you start off by implementing a small set of the software requirements. These are then enhanced iteratively in the evolving versions until the system is completed. This process model starts with part of the software, which is then implemented and reviewed to identify further requirements.
Like the incremental model, the iterative model allows you to see the results at the early stages of development. This makes it easy to identify and fix any functional or design flaws. It also makes it easier to manage risk and change requirements.

The deadline and budget may change throughout the development process, especially for large complex projects. The iterative model is a good choice for large software that can be easily broken down into modules.

Specialized Process Models

Specialized process models use many of the characteristics of one or more of the conventional models presented so far, however they tend to be applied when a narrowly defined software engineering approach is chosen. They include, Components based development.

Component Based Development The component based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, Specialized process model demanding an iterative approach to the creation of software.However, the component based development model constructs applications from prepackaged software components.Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object oriented classes or packages of classes.Regardless of the technology that is used to create the components, the component based development specialized process model incorporates the following steps.

1. Available component based products are researched and evaluated for the application
   domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The component based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.

Spiral Model

The spiral model is a risk driven iterative software process model. The spiral model delivers projects in loops. Unlike other process models, its steps aren't activities but phases for addressing whatever problem has the greatest risk of causing a failure. It was designed to include the best features from the waterfall and introduces risk-assessment.

You have the following phases for each cycle:

Address the highest-risk problem and determine the objective and alternate solutions
Evaluate the alternatives and identify the risks involved and possible solutions
Develop a solution and verify if it's acceptable
Plan for the next cycle

You develop the concept in the first few cycles, and then it evolves into an implementation. Though this model is great for managing uncertainty, it can be difficult to have stable documentation. The spiral model can be used for projects with unclear needs or projects still in research and development.

•Each phase of the Spiral Model is divided into four quadrants as shown in the above figure. The functions of these four quadrants are discussed below-

•**Objectives determination and identify alternative solutions:** Requirements are gathered from the customers and the objectives are identified, elaborated, and analyzed at the start of every phase. Then alternative solutions possible for the phase are proposed in this quadrant.

•**Identify and resolve Risks:** During the second quadrant, all the possible solutions are evaluated to select the best possible solution. Then the risks associated with that solution are identified and the risks are resolved using the best possible strategy. At the end of this quadrant, the Prototype is built for the best possible solution.

•**Develop next version of the Product:** During the third quadrant, the identified features are developed and verified through testing. At the end of the third quadrant, the next version of the software is available.

•**Review and plan for the next Phase:** In the fourth quadrant, the Customers evaluate the so far developed version of the software. In the end, planning for the next phase is started.

Risk Handling in Spiral Model

A risk is any adverse situation that might affect the successful completion of a software project. The most important feature of the spiral model is handling these unknown risks after the project has started. Such risk resolutions are easier done by developing a prototype. The spiral model supports copying up with risks by providing the scope to build a prototype at every phase of the software development.

Unified process model

The unified process model (or UPM is an iterative, incremental, architecture-centric, and use-case driven approach for developing software. This model consists of four phases, including: Inception, in which you collect requirements from the customer and analyze the project's feasibility, its cost, risks, and profits.

The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but characterize them in a way that implements many of the best principles of agile software development.

The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer’s view of a system. It emphasizes the important role of software architecture and “helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse”.

It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

Comparison of different life cycle models

Classical waterfall model

Basic model and all other life cycle models as embellishments of this model.
Can not be used in practical development projects
No mechanism to handle the errors committed during any of the phases.

Iterative waterfall model.

Most widely used.
Simple to understand and use.
Suitable only for well-understood problems;
Not suitable for very large projects and for projects that are subject to many risks.

Prototyping model

suitable for projects for which either the user requirements or the underlying technical aspects are not well understood.
especially popular for development of the user-interface part of the projects.

Evolutionary Model

suitable for large problems which can be decomposed into a set of modules for incremental development and delivery.
used for object-oriented development projects.

Spiral model

meta model since it encompasses all other life cycle models.
Risk handling is inherently built into this model.
suitable for development of technically challenging software products that are prone to several kinds of risks.
much more complex than the other models

Software Requirements(SRS)

Software Requirement Specification (SRS) Format as name suggests, is complete specification and description of requirements of software that needs to be fulfilled for successful development of software system. These requirements can be functional as well as non-functional depending upon type of requirement. The interaction between different customers and contractor is done because its necessary to fully understand needs of customers.

SRS document is one of the most critical documents in software development.
It describes how a software system should be developed. Simply put, an SRS provides everyone involved with a roadmap for that project.
An SRS document forces you to put the idea down on paper to cover all these details. You must translate this idea into a language that developers understand.
An SRS document describes what a client wants and what developers will provide. It is the written agreement on every detail of the app.
Designers get project insights through SRS documents so they can match the design to the use case.
Testers get the guidelines for creating test cases that match the business’s needs.

End-users use the SRS to understand the software.
It provides investors with an overview of the system’s features so they can make investment decisions.
An SRS is important because it is a single source of information and expectations, which prevents misunderstandings between project managers, developers, designers, and testers.

What does an SRS include?

1.The purpose of the software being developed
2.An overall description of the software
3.The functionality of the software or what it is supposed to do
4.Performance of the software in a production situation
5.Non-functional requirements
6.External interfaces or how the software will interact with hardware or other software it must connect to
7.Design constraints or the limitations of the environment that the software will run in

Differences between Functional and Non Functional Requirements

Functional Requirements	Non-functional requirements
Functional requirements help to understand the functions of the system.	They help to understand the system's performance.
Functional requirements are mandatory.	While non-functional requirements are not mandatory.
They are easy to define.	They are hard to define.
They describe what the product does.	They describe the working of product.
It concentrates on the user's requirement.	It concentrates on the expectation and experience of the user.
It helps us to verify the software's functionality.	It helps us to verify the software's performance.
These requirements are specified by the user.	These requirements are specified by the software developers, architects, and technical persons.

There is functional testing such as API testing, system, integration, etc.	There is non-functional testing such as usability, performance, stress, security, etc.
Examples of the functional requirements are - Authentication of a user on trying to log in to the system.	Examples of the non-functional requirements are - The background color of the screens should be light blue.
These requirements are important to system operation.	These are not always the important requirements, they may be desirable.
Completion of Functional requirements allows the system to perform, irrespective of meeting the non-functional requirements.	While system will not work only with non-functional requirements.

User Interface requirements

UI is an important part of any software or hardware or hybrid system. A software is widely accepted if it is -

easy to operate, quick in response, effectively handling operational errors, providing simple yet consistent user interface

User acceptance majorly depends upon how user can use the software. UI is the only way for users to perceive the system. A well performing software system must also be equipped with attractive, clear, consistent and responsive user interface. Otherwise the functionalities of software system can not be used in convenient way. A system is said be good if it provides means to use it efficiently. User interface requirements are briefly mentioned below -

Content presentation, Easy Navigation,Simple interface, Responsive, Consistent UI elements, Feedback mechanism, Default settings, Purposeful layout, Strategical use of color and texture, Provide help information, User centric approach, Group based view settings.

System Requirements

What Does System Requirements Mean?

System requirements are the configuration that a system must have in order for a hardware or software application to run smoothly and efficiently. Failure to meet these requirements can result in installation problems or performance problems. The former may prevent a device or application from getting installed, whereas the latter may cause a product to malfunction or perform below expectation or even to hang or crash.System requirements are also known as minimum system requirements.

Techopedia Explains System Requirements
For packaged products, system requirements are often printed on the packaging. For downloadable products, the system requirements are often indicated on the download page. System requirements can be broadly classified as functional requirements, data requirements, quality requirements and constraints. They are often provided to consumers in complete detail. System requirements often indicate the minimum and the recommended configuration. The former is the most basic requirement, enough for a product to install or run, but performance is not guaranteed to be optimal. The latter ensures a smooth operation.

Hardware system requirements often specify the operating system version, processor type, memory size, available disk space and additional peripherals, if any, needed. Software system requirements, in addition to the aforementioned requirements, may also specify additional software dependencies (e.g., libraries, driver version, framework version). Some hardware/software manufacturers provide an upgrade assistant program that users can download and run to determine whether their system meets a product's requirements.

Software Requirement Specification

SRS is a document created by system analyst after the requirements are collected from various stakeholders.
SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.
The requirements received from client are written in natural language. It is the responsibility of system analyst to document the requirements in technical language so that they can be comprehended and useful by the software development team.
SRS should come up with following features:
User Requirements are expressed in natural language. Technical requirements are expressed in structured language, which is used inside the organization. Design description should be written in Pseudo code. Format of Forms and GUI screen prints. Conditional and mathematical notations for DFDs etc.

Software Requirement Validation

After requirement specifications are developed, the requirements mentioned in this document are validated. User might ask for illegal, impractical solution or experts may interpret the requirements incorrectly. This results in huge increase in cost if not nipped in the bud. Requirements can be checked against following conditions -
If they can be practically implemented, If they are valid and as per functionality and domain of software, If there are any ambiguities, If they are complete, If they can be demonstrated

Software Requirements Characteristics

Gathering software requirements is the foundation of the entire software development project. Hence they must be clear, correct and well-defined.
A complete Software Requirement Specifications must be:
● Clear, Correct, Consistent, Coherent, Comprehensible, Modifiable, Verifiable, Prioritized, Unambiguous, Traceable, Credible source, Complete

Requirement Elicitation Process

Requirement elicitation process can be depicted using the folloiwng diagram:
●Requirements gathering - The developers discuss with the client and end users and know their expectations from the software.
●Organizing Requirements - The developers prioritize and arrange the requirements in order of importance, urgency and convenience.
●Negotiation & discussion - If requirements are ambiguous or there are some conflicts in requirements of various stakeholders, if they are, it is then negotiated and discussed with stakeholders. Requirements may then be prioritized and reasonably compromised.
The requirements come from various stakeholders. To remove the ambiguity and conflicts, they are discussed for clarity and correctness. Unrealistic requirements are compromised reasonably.
●Documentation - All formal & informal, functional and non-functional requirements are documented and made available for next phase processing.

Technique
- Interviews, Questionnaires, Task analysis, Domain Analysis, Brainstorming Prototyping, Observation

Requirements Validation Techniques

Requirements validation is the process of checking that requirements defined for development, define the system that the customer really wants. To check issues related to requirements, we perform requirements validation. We usually use requirements validation to check error at the initial phase of development as the error may increase excessive rework when detected later in the development process.

In the requirements validation process, we perform a different type of test to check the requirements mentioned in the Software Requirements Specification (SRS), these checks include:
- Completeness checks, Consistency checks, Validity checks, Realism checks, Ambiguity checks, Verifiability

The output of requirements validation is the list of problems and agreed on actions of detected problems. The lists of problems indicate the problem detected during the

process of requirement validation. The list of agreed action states the corrective action that should be taken to fix the detected problem.

There are several techniques which are used either individually or in conjunction with other techniques to check to check entire or part of the system:

1.**Test case generation**:
Requirement mentioned in SRS document should be testable, the conducted tests reveal the error present in the requirement. It is generally believed that if the test is difficult or impossible to design than, this usually means that requirement will be difficult to implement and it should be reconsidered.

2.**Prototyping:**
In this validation techniques the prototype of the system is presented before the end-user or customer, they experiment with the presented model and check if it meets their need. This type of model is generally used to collect feedback about the requirement of the user.

3.**Requirements Reviews:**
In this approach, the SRS is carefully reviewed by a group of people including people from both the contractor organisations and the client side, the reviewer systematically analyses the document to check error and ambiguity.

4.**Automated Consistency Analysis:**
This approach is used for automatic detection of an error, such as nondeterminism, missing cases, a type error, and circular definitions, in requirements specifications. First, the requirement is structured in formal notation then CASE tool is used to check in-consistency of the system, The report of all inconsistencies is identified and corrective actions are taken.

5.**Walk-through:**
A walkthrough does not have a formally defined procedure and does not require a differentiated role assignment.

1.Checking early whether the idea is feasible or not.
2.Obtaining the opinions and suggestion of other people.
3.Checking the approval of others and reaching an agreement.

System Modeling

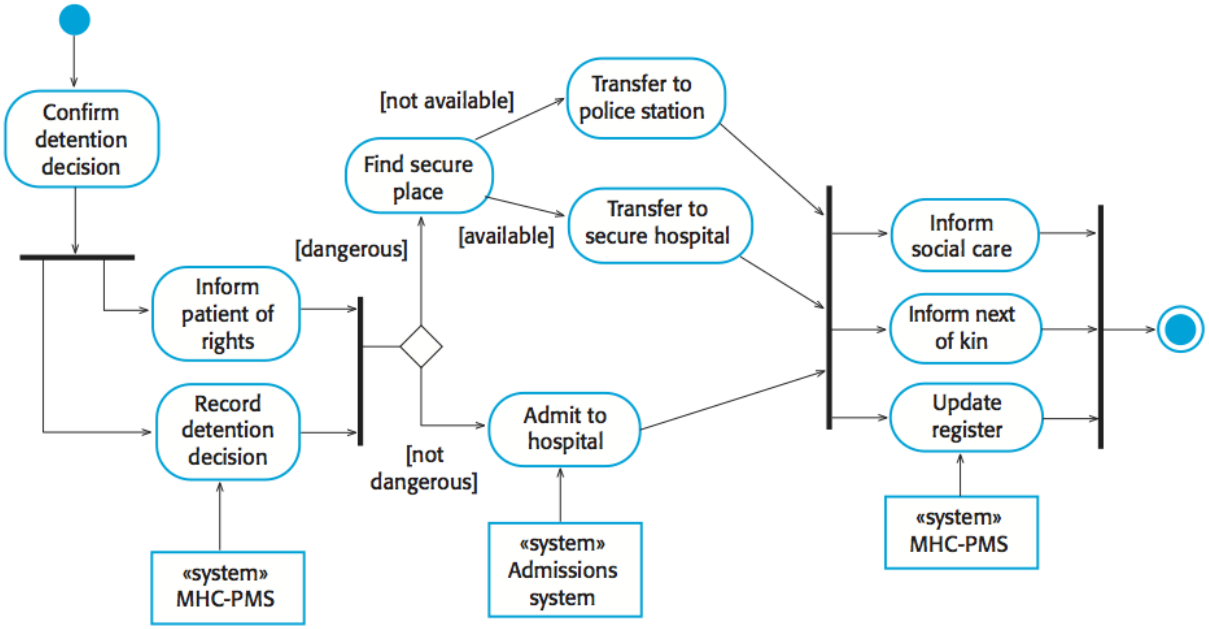
System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. It is about representing a system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML). Models help the analyst to understand the functionality of the system; they are used to communicate with customers.

Models can explain the system from different perspectives:

●An external perspective, where you model the context or environment of the system.

●An interaction perspective, where you model the interactions between a system and its environment, or between the components of a system.
●A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
●A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.
Five types of UML diagrams that are the most useful for system modeling:
●Activity diagrams, which show the activities involved in a process or in data processing.
●Use case diagrams, which show the interactions between a system and its environment.
●Sequence diagrams, which show interactions between actors and the system and between system components.
●Class diagrams, which show the object classes in the system and the associations between these classes.
●State diagrams, which show how the system reacts to internal and external events.

Context and process models
Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries. Social and organizational concerns may affect the decision on where to position system boundaries. Architectural models show the system and its relationship with other systems.
System boundaries are established to define what is inside and what is outside the system. They show other systems that are used or depend on the system being developed. The position of the system boundary has a profound effect on the system requirements. Defining a system boundary is a political judgment since there may be pressures to develop system boundaries that increase/decrease the influence or workload of different parts of an organization.
Context models simply show the other systems in the environment, not how the system being developed is used in that environment. Process models reveal how the system being developed is used in broader business processes. UML activity diagrams may be used to define business process models.
The example below shows a UML activity diagram describing the process of involuntary detention and the role of MHC-PMS (mental healthcare patient management system) in it.



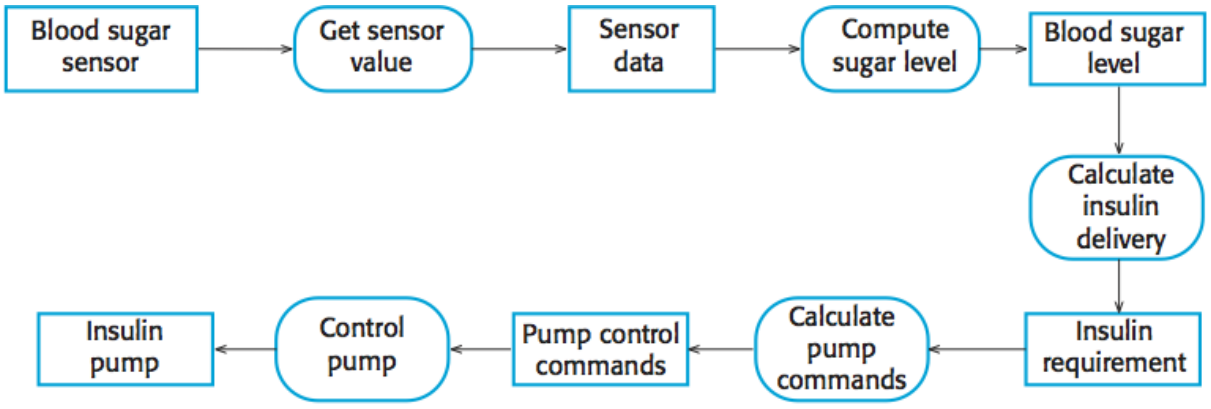
Behavioral models

Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment. Two types of stimuli:

●Some data arrives that has to be processed by the system.

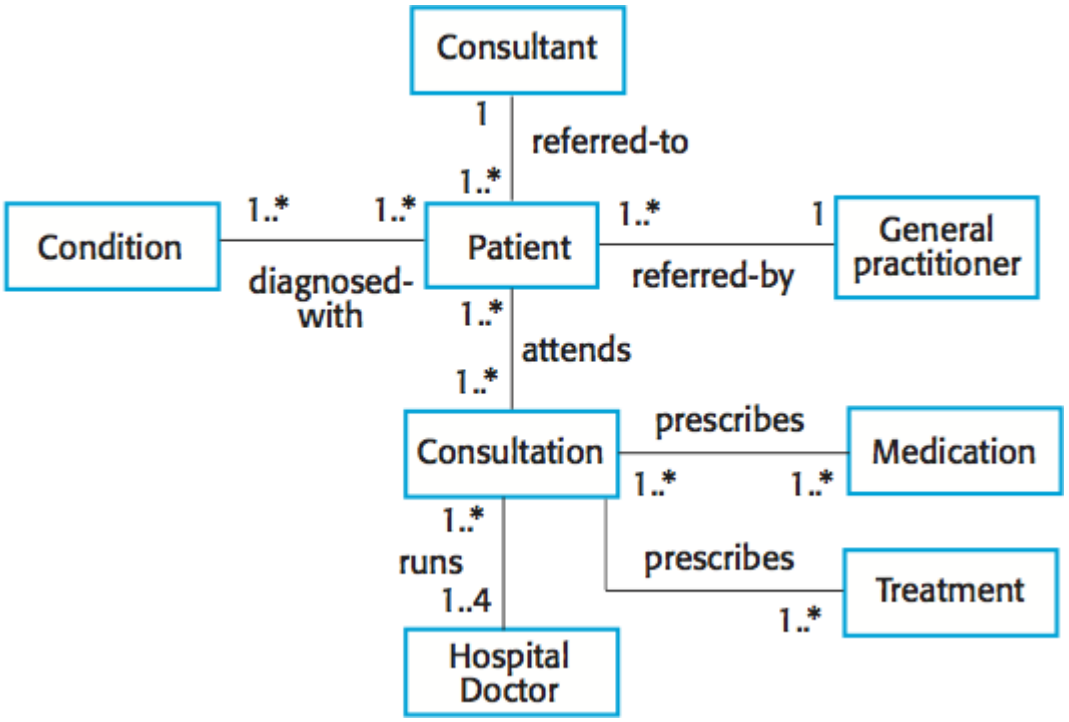
●Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing. Data-driven models show the sequence of actions involved in processing input data and generating an associated output. They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system. Data-driven models can be created using UML activity diagrams:



Structural models

Structural models of software display the organization of a system in terms of the components that make up that system and their relationships. Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing. You create structural models of a system when you are discussing and designing the system architecture. UML class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes. An object class can be thought of as a general definition of one kind of system object. An association is a link between classes that indicates that there is some relationship between these classes. When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.

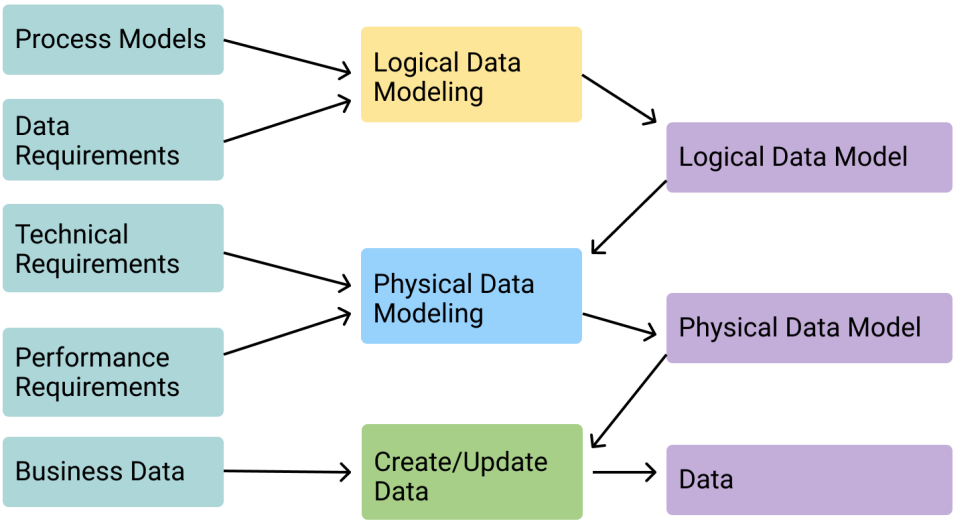


Data Modeling

The process of analyzing data objects and their relationships with other objects is known as Data Modeling. It's used to look into the data requirements for various business processes. The Data Models are created to store the information in a database. Instead of focusing on what operations you need to perform, the Data Model focuses on what data is required and how you need to organize it.

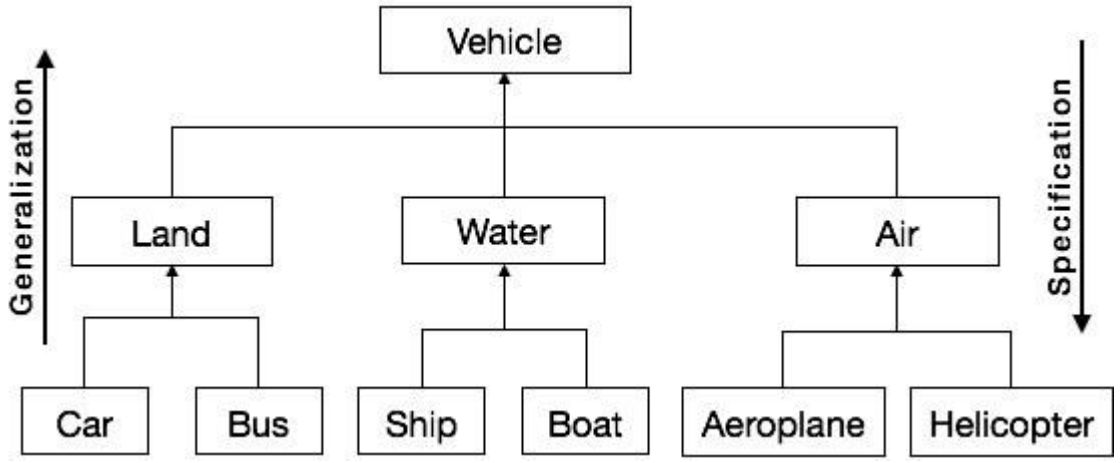
Data Modeling is the process of creating a Data Model using formal Data Model descriptions and Data Modeling techniques in Software Engineering. A Data Model is also known as Database Modeling because it is eventually implemented in a database.

This article explains the Data Modeling Concepts in Software Engineering including types of Data Models, Data Modeling tools, and the need for a Data Model.



Object model

An object model is a logical interface, software or system that is modeled through the use of object-oriented techniques. It enables the creation of an architectural software or system model prior to development or programming. An object model is part of the object-oriented programming (OOP) lifecycle.



Software Design

The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language.

The software design process can be divided into the following three levels of phases of design:

Interface Design, Architectural Design, Detailed Design

Interface Design:
Interface design is the specification of the interaction between a system and its environment. this phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during interface design, the internal of the systems are completely ignored and the system is treated as a black box. Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts. The design problem statement produced during the problem analysis step should identify the people, other systems, and devices which are collectively called agents.

Architectural Design:
Architectural design is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them. In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored.

Detailed Design:
Design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures.

Software Quality

Software quality product is defined in term of its fitness of purpose. That is, a quality product does precisely what the users want it to do. For software products, the fitness of use is generally explained in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory interpretation of quality for many devices such as a car, a table fan, a grinding machine, etc.for software products, "fitness of purpose" is not a wholly satisfactory definition of quality.

Example: Consider a functionally correct software product. That is, it performs all tasks as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally right, we cannot consider it to be a quality product.

The modern view of a quality associated with a software product several quality methods such as the following:

Portability: A software device is said to be portable, if it can be freely made to work in various operating system environments, in multiple machines, with other software products, etc.

Usability: A software product has better usability if various categories of users can easily invoke the functions of the product.

Reusability: A software product has excellent reusability if different modules of the product can quickly be reused to develop new products.

Correctness: A software product is correct if various requirements as specified in the SRS document have been correctly implemented.

Maintainability: A software product is maintainable if bugs can be easily corrected as and when they show up, new tasks can be easily added to the product, and the functionalities of the product can be easily modified, etc.

Software Designing Concept

Software design principles are concerned with providing means to handle the complexity of the design process effectively. Effectively managing the complexity will not only reduce the effort needed for design but can also reduce the scope of introducing errors during design.

The following points should be considered while designing Software:
Abstraction- hide Irrelevant data
Abstraction simply means to hide the details to reduce complexity and increases efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broad ways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.
Modularity- subdivide the system
Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity in design has now become a trend and is also important. If the system contains fewer components then it would mean the system is complex which requires a lot of effort (cost) but if we are able to divide the system into components then the cost would be small.
Architecture- design a structure of something
Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and

the data of the structure. These components interact with each other and use the data of the structure in architecture.

Refinement- removes impurities

Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is actually a process of developing or presenting the software or system in a detailed manner that means to elaborate a system or software. Refinement is very necessary to find out any error if present and then to reduce it.

Pattern- a repeated form

The pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.

Information Hiding- hide the information

Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can't be accessed by any other modules.

Refactoring- reconstruct something

Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without affecting the behavior or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't affect the behavior of the design and improves the internal structure".

Architectural Design

The software needs the architectural design to represents the design of software. IEEE defines architectural design as "the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system." The software that is built for computer-based systems can exhibit one of these many architectural styles.

Each style will describe a system category that consists of :

A set of components(eg: a database, computational modules) that will perform a function required by the system.

The set of connectors will help in coordination, communication, and cooperation between the components.

Conditions that how components can be integrated to form the system.

Semantic models that help the designer to understand the overall properties of the system.

The use of architectural styles is to establish a structure for all the components of the system.

Software Architecture

Architecture serves as a blueprint for a system. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.
It defines a structured solution to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.
Further, it involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product.
These decisions comprise of −
Selection of structural elements and their interfaces by which the system is composed.
Behavior as specified in collaborations among those elements.
Composition of these structural and behavioral elements into large subsystem.
Architectural decisions align with business objectives.
Architectural styles guide the organization.

Data Design

The data design action translates data defined as part of the analysis model into data structures at the software component level and. When necessary into a database architecture at the application level.
The data structures needed for implementing the software as well-as the operations that can be applied on them should be identified.
A data dictionary should be developed to depict how different data objects interact with each other and what constraints are to be imposed on the elements of data structure.
Stepwise refinement should be used in data design process and detailed design decisions should be made later in the process.
Only those modules that need to access data stored in a data structure directly should be aware of the representation of the data structure.
A library containing the set of useful data structures along with the operations that can be performed on them should be maintained.
Language used for developing the system should support abstract data types.

Architectural Style

The architectural style shows how do we organize our code, or how the system will look like from 10000 feet helicopter view to show the highest level of abstraction of our system design. Furthermore, when building the architectural style of our system we focus on layers and modules and how they are communicating with each other.
There are different types of architectural styles, and moreover, we can mix them and produce a hybrid style that consists of a mix between two and even more

architectural styles. Below is a list of architectural styles and examples for each category:
Structure architectural styles: such as layered, pipes and filters and component-based styles.
Messaging styles: such as Implicit invocation, asynchronous messaging and publish-subscribe style.
Distributed systems: such as service-oriented, peer to peer style, object request broker, and cloud computing styles.
Shared memory styles: such as role-based, blackboard, database-centric styles.
Adaptive system styles: such as microkernel style, reflection, domain-specific language styles.

Architectural Patterns
The architectural pattern shows how a solution can be used to solve a reoccurring problem. In another word, it reflects how a code or components interact with each other. Moreover, the architectural pattern is describing the architectural style of our system and provides solutions for the issues in our architectural style. Personally, I prefer to define architectural patterns as a way to implement our architectural style. For example: how to separate the UI of the data module in our architectural style? How to integrate a third-party component with our system? how many tires will we have in our client-server architecture? Examples of architectural patterns are microservices, message bus, service requester/ consumer, MVC, MVVM, microkernel, n-tier, domain-driven design, and presentation-abstraction-control.

Design patterns
Design patterns are accumulative best practices and experiences that software professionals used over the years to solve the general problem by – trial and error – they faced during software development. The Gang of Four (GOF, refers to Eric Gamma, Richard Helm, Ralf Johnson, and John Vlissides) wrote a book in 1994 titled with “Design Pattern – Elements of reusable object-oriented software” in which they suggested that design patterns are based on two main principles of object-oriented design:
Develop to an interface, not to an implementation.
Favor object composition over inheritance.

Class-Based Component Design
Class-based component design is a method for designing software components. It uses a class or classes (a collection of related data items, and the operations needed to manipulate those items) to represent the component in question. Consider for a moment the email application on your personal computer. One component of this system might be an email message. If we define a class called 'EmailMessage', then a data item might be 'message', and an operation that can be performed on 'message' might be 'SendMessage'. It's likely that this will have more data and operations defined, but you get the idea.

Functional Independence

Functional independence is the refined form of the design concepts of modularity, abstraction, and information hiding. Functional independence is achieved by developing a module in such a way that it uniquely performs given sets of function without interacting with other parts of the system. The software that uses the property of functional independence is easier to develop because its functions can be categorized in a systematic manner. Moreover, independent modules require less maintenance and testing activity, as secondary effects caused by design modification are limited with less propagation of errors. In short, it can be said that functional independence is the key to a good software design and a good design results in high-quality software. There exist two qualitative criteria for measuring functional independence, namely, coupling and cohesion.

Coupling

Coupling measures the degree of interdependence among the modules. Several factors like interface complexity, type of data that pass across the interface, type of communication, number of interfaces per module, etc. influence the strength of coupling between two modules. For better interface and well-structured system, the modules should be loosely coupled in order to minimize the 'ripple effect' in which modifications in one module results in errors in other modules. Module coupling is categorized into the following types.

No direct coupling: Two modules are said to be 'no direct coupled' if they are independent of each other.
Data coupling: Two modules are said to be 'data coupled' if they use parameter list to pass data items for communication.
Stamp coupling: Two modules are said to be 'stamp coupled' if they communicate by passing a data structure that stores additional information than what is required to perform their functions.
Control coupling: Two modules are said to be 'control coupled' if they communicate (pass a piece of information intended to control the internal logic) using at least one 'control flag'. The control flag is a variable whose value is used by the dependent modules to make decisions.

Cohesion

Cohesion measures the relative functional strength of a module. It represents the strength of bond between the internal elements of the modules. The tighter the elements are bound to each other, the higher will be the cohesion of a module. In practice, designers should avoid a low level of cohesion when designing a module. Generally, low coupling results in high cohesion and vice versa.

Various types of cohesion are listed below.

Functional cohesion: In this, the elements within the modules are concerned with the execution of a single function.

Sequential cohesion: In this, the elements within the modules are involved in activities in such a way that the output from one activity becomes the input for the next activity.

Communicational cohesion: In this, the elements within the modules perform different functions, yet each function references the same input or output information.

Procedural cohesion: In this, the elements within the modules are involved in different and possibly unrelated activities.

Temporal cohesion: In this, the elements within the modules contain unrelated activities that can be carried out at the same time.

Logical cohesion: In this, the elements within the modules perform similar activities, which are executed from outside the module.

Coincidental cohesion: In this, the elements within the modules perform activities with no meaningful relationship to one another.

Conducting component level

Component-level design is elaborative in nature.It transforms information from requirements and architectural models into a design representation that provides sufficient detail to guide the construction (coding and testing) activity. The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain.
Using the requirements and architectural model, each analysis class and architectural component is elaborated…

Step 2. Identify all design classes that correspond to the infrastructure domain.
These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point.
Classes and components in this category include,GUI components (often available as reusable components), Operating system components, Object and data management components

Step 3. Elaborate all design classes that are not acquired (Obtain) as reusable components.Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling).
In general, data structures and types used to define attributes are defined within the context of the programming language that is to be used for implementation.
UML defines an attribute's data type using the following syntax:

Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.

Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design.
Step 5. Develop and elaborate behavioral representations for a class or component. UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system.
Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams are used as part of architectural.
Step 7. Refactor every component-level design representation and always consider alternatives.

Object Constraint Language
The Object Constraint Language (OCL) is an expression language that describes constraints on object-oriented languages and other modelling artifacts. A constraint can be seen as a restriction on a model or a system. OCL is part of Unified Modeling Language (UML) and it plays an important role in the analysis phase of the software lifecycle.
Object Constraint Language (OCL), is a formal language
to express side effect-free constraints. Users of the Unified Modeling Language and other
languages can use OCL to specify constraints and other expressions attached to their models.

OCL is the expression language for the Unified Modeling Language (UML). To understand OCL, the component parts of this statement should be examined. Thus, OCL has the characteristics of an expression language, a modeling language and a formal language.

- Expression language

OCL is a pure expression language. Therefore, an OCL expression is guaranteed to be without side effect. It cannot change anything in the model. This means that the state of the system will never change because of an OCL expression, even though an OCL expression can be used to specify such a state change (e.g., in a post-condition). All values for all objects, including all links, will not change. Whenever an OCL expression is evaluated, it simply delivers a value.

- Modeling language

OCL is a modeling language, not a programming language. It is not possible to write program logic or flow-control in OCL. You especially cannot invoke processes or activate non-query operations within OCL. Because OCL is a modeling language in the first place, not everything in it is promised to be directly executable.
As a modeling language, all implementation issues are out of scope and cannot be expressed in OCL. Each OCL expression is conceptually atomic. The state of the objects in the system cannot change during evaluation.

- Formal language

OCL is a formal language where all constructs have a formally defined meaning. The specification of OCL is part of the UML specification.OCL is not intended to replace existing formal languages, like VDM, Z etc

User Interface Design

The visual part of a computer application or operating system through which a client interacts with a computer or software. It determines how commands are given to the computer or the program and how data is displayed on the screen.

Types of User Interface

There are two main types of User Interface:

- Text-Based User Interface or Command Line Interface
- Graphical User Interface (GUI)

Text-Based User Interface: This method relies primarily on the keyboard. A typical example of this is UNIX.

Advantages

- Many and easier to customizations options.
- Typically capable of more important tasks.

Disadvantages

- Relies heavily on recall rather than recognition.
- Navigation is often more difficult.

Graphical User Interface (GUI): GUI relies much more heavily on the mouse. A typical example of this type of interface is any versions of the Windows operating systems.

Advantages

- Less expert knowledge is required to use it.
- Easier to Navigate and can look through folders quickly in a guess and check manner.
- The user may switch quickly from one task to another and can interact with several different applications.

Disadvantages

- Typically decreased options.
- Usually less customizable. Not easy to use one button for tons of different variations.

UI Design Principles

Structure: Design should organize the user interface purposefully, in the meaningful and usual based on precise, consistent models that are apparent and recognizable to users, putting related things together and separating unrelated things, differentiating dissimilar things and making similar things resemble one another. The structure principle is concerned with overall user interface architecture.

Simplicity: The design should make the simple, common task easy, communicating clearly and directly in the user's language, and providing good shortcuts that are meaningfully related to longer procedures.

Visibility: The design should make all required options and materials for a given function visible without distracting the user with extraneous or redundant data.
Feedback: The design should keep users informed of actions or interpretation, changes of state or condition, and bugs or exceptions that are relevant and of interest to the user through clear, concise, and unambiguous language familiar to users.
Tolerance: The design should be flexible and tolerant, decreasing the cost of errors and misuse by allowing undoing and redoing while also preventing bugs wherever possible by tolerating varied inputs and sequences and by interpreting all reasonable actions.

Design evolution

Design evolution is a simplified form of network optimisation, since the basic structure of the network is maintained while smaller changes are made to the network by removing small heat exchangers. As in any optimisation, design evolution also needs degrees of freedom