B.C.A study

Unit- 1 :Software Engineering

Definition –

Software engineering is the process of analyzing user needs and designing, constructing, and testing end user applications that will satisfy these needs through the use of software programming languages. It is the application of engineering principles to software development. In contrast to simple programming, software engineering is used for larger and more complex software systems, which are used as critical systems for businesses and organizations.

A software engineer takes the software needs of end users into account and consequently develops or designs new applications. Furthermore, software engineering may involve the process of analyzing existing software and modifying it to meet current application needs.

As computer hardware becomes cheaper, the focus transfers to software systems. Large software systems may be more complex than the hardware used to run them, so there is great demand for best practices and engineering processes that can be applied to software development. There must be discipline and control during software engineering, much like any complex engineering endeavor.

In modern consumer electronics, devices in direct competition often have similar hardware and processing power, but the user experience will vary greatly depending on the software being used.

History of Software Paradigms

Since the development of the early projects of large software systems in the 1950's and 1960's computer scientists started to develop Software Development Life Cycle (SDLC) models to govern the whole process of development. A number of process models have been developed over the years. The purpose of these models was to be used as a guide and a conceptual scheme to manage the system software project in a systematic, formal and procedural way. They also provide a guide for the development of software applications. Software models are used to guide the various tasks of the software system project including planning, organizing, staffing, coordinating, budgeting, and directing software development activities.

Introduction

“Paradigm”, a Greek word meaning “example”, is commonly used to refer to a category of entities that share a common characteristic. Software engineering paradigms are also known as Software engineering models or Software Development Models.

In order to reduce the potential chaos of developing software applications and systems, we use software process models and paradigms that describe the tasks that are required for the building of high-quality software systems. The specific process model or paradigms used to develop a given system depends heavily on the nature of the target system. Use of software paradigms in the development of the software processes has many benefits, including supporting systematic approach and the use of standard approaches and methodologies.

The software engineering paradigm which is also referred to as a software process model or Software Development Life Cycle (SDLC) model is the development strategy that encompasses the process, methods and tools. SDLC describes the period of time that starts with the software system being conceptualized and ends with the software system been discarded after usage.

The software engineering paradigm provides the guidance to the software engineer.

A paradigm specifies the particular approach or philosophy for designing, building and maintaining software. Each paradigm has its own advantages and disadvantages which make some paradigm more suitable to be used in developing a given software system in a given situation than another.

There are common software process tasks, phases and activities that are modeled by software models. They are heavily affected by selected software paradigms. These tasks, phases and activities include:

Requirements Engineering: Software specification and functional requirements obtained from the user.

Requirements Analysis and Modeling

Architectural Engineering, implementation and Design: Production of the software system as a product

Software Testing and Validation: Activity that assures that customer specifications are met

System Delivery.

Software Evolution and Upgrading: System modification to meet continuing customer needs

System Documenting

Maintenance

The Software Engineering Paradigms (Process Models)

The four basic models are:

Waterfall model, also known as the traditional software development life cycle (SDLC).

The spiral model

Incremental process model

Agile development model

Waterfall Model

The waterfall model is also known as the linear sequential model and as classic life cycle model.

This model was developed in 1970 to make the software development process more structured. It is the oldest and most commonly used software engineering.

It demands a systematic, sequential approach to software development that begins at the requirements analysis through planning, design, coding, testing, and delivery.

The waterfall model, drawn from an engineering model is used in order to control the development of large software systems. The waterfall model is visualized as the flowchart in Figure 1. It consists of different stages which are processed in a linear fashion. The steps in the waterfall mode are: requirements analysis, planning, design, coding, testing, and delivery. Brief description of the phases in the waterfall model is presented below.

Requirements Analysis

In this phase the requirements of the software system are defined. This answers the 'What is the software system to be developed' question. It involves extensive user participation and ends with an approved set of requirements documented in a software requirement specification (SRS) document. This document is the basis of next stages of the project.

Design

In this phase the logical design of the system is defined. This answers the 'How is the software system will be developed' question. This phase uses the requirements specified in the previous phase (the SRS document) and translates them into design which will solve the problem. The design phase is an intermediate phase and bridge between 'What' the user/customer wants and the implemented system (code) that will be created to satisfy the requirements.

Coding

This stage of the model involves the writing of the code. The actual design is turned into a set of programs.

Testing

The code that is developed in the implementation phase is tested during the testing phase. This involves unit testing for the lowest level components, integration testing for groups of components and testing of the system as a whole.

A part of the testing phase are the verification and validation of the system for acceptance (acceptance testing). Verification checks that the system is correct (building the system right) while validation checks that the system meets the users desires (building the right system). System validation fulfils the use needs and and requires considerable user involvement.

Delivery

In this phase the system is made operational. Main activities in this phase include training of the user's, and installation of the system. Also, in this phase the system is maintained where bugs, errors, and defects are corrected. The system is made more efficient, new requirements are added and existing functionality and features modified to meet the changing customer/user/business needs.

The Spiral Model

The spiral process model (developed by Barry Boehm in 1989) includes the same steps (requirements analysis, planning, design, coding, testing, and delivery) as the waterfall process model.

The motivation behind the development of the spiral process model was due to fact that requirements given at the beginning of the project are incomplete where requirements analysis phase and the design phase are continuously evolving as time passes. The steps go through a cyclical motion as requirements and design

The spiral model focuses on the constant re-evaluation of the design and risks involved. It can be described as being 'iterative' and 'sequential'. The model can accommodate any process development model due to its generality. It is used commonly used for large and complex projects which as a result, can also be described as being expensive.

Incremental Process Model

The incremental process model allows customers to gent the developed system incrementally and part by part. They do not need to wait for the whole system to be completed. This is especially important in very large system where the completion of the whole system takes very long time. Deliverables are produced by increments where after an initial information strategy planning, the customer could get the first deliverable which is the core product. Here, the first 'increment' is the core product or the most important functionality of the system. Basic requirements are addressed, and the core product is produced. Non-basic requirements (some known, others unknown) are addressed later and the rest of the project remains undelivered. The core product is used and reviewed by the customer and base on customer evaluation, a plan is developed for the next increment. The plan addresses the new requirements specified by the customer of the core product to better meet the needs of the customer and the delivery of additional features and functionality. The incremental process is repeated following the delivery of each increment, until complete product is produced. The incremental process model is both evolutionary and interactive approach

Agile Development Model

The Agile development model, also known as the "Extreme Programming" model has the sole focus of time of delivery of the project. It is designed to allow software projects to be completed in the shortest time possible. Hence, agile development models are appropriate for web applications developers, which require rapid delivery of the system.

The incremental process model has the following features:

There are many agile development methods; most minimize risk by developing software in short time.

One iteration in agile development model is the Software developed in one unit of time.

Each iteration is an entire software project includes all stages that exist in the model which are planning, requirements analysis, design, coding, testing, and documentation.

An iteration does not necessarily add enough functionality to guarantee releasing the product to market. But, at the end of the iteration, there should exists an available release (without bugs).

At the end of each iteration, the development team re-evaluates project priorities.

Agile methods emphasize face-to-face communication over written documents.

A Generic View of Software Engineering

Engineering is the analysis, design, construction, verification, and management of technical (or social) entities. Regardless of the entity to be engineered, the following questions must be asked and answered:
• What is the problem to be solved?
• What characteristics of the entity are used to solve the problem?
• How will the entity (and the solution) be realized?
• How will the entity be constructed?
• What approach will be used to uncover errors that were made in the design
and construction of the entity?
• How will the entity be supported over the long term, when corrections, adaptations,
and enhancements are requested by users of the entity.

The work associated with software engineering can be categorized into three generic phases, regardless of application area, project size, or complexity. Each phase addresses one or more of the questions noted previously.

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. The key requirements of the system and the software are identified. Although the methods applied during the definition phase will vary depending on the software engineering paradigm (or combination of paradigms) that is applied, three major tasks will occur in some form: system or information engineering, software project planning, and requirements analysis .

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 procedural details are to be implemented, how interfaces are to be characterized, how the design will be translated into a programming language (or nonprocedural language), and how testing will be performed. The methods applied during the development phase will vary, but three specific technical tasks should always occur: software design, code generation,and software testing.

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. The support phase reapplies the steps of the definition and development phases but does so in the context of existing software. Four types of change are encountered during the support phase:

Correction. Even with the best quality assurance activities, it is likely that the customer will uncover defects in the software. Corrective maintenance changes the software to correct defects.
Adaptation. Over time, the original environment (e.g., CPU, operating system, business rules, external product characteristics) for which the software was developed is likely to change. Adaptive maintenance results in modification to the software to accommodate changes to its external environment.
Enhancement. As software is used, the customer/user will recognize additional functions that will provide benefit. Perfective maintenance extends the software beyond its original functional requirements.
Prevention. Computer software deteriorates due to change, and because of this, preventive maintenance, often called software reengineering, must be conducted to enable the software to serve the needs of its end users. In essence, preventive maintenance makes changes to computer programs so that they can be more easily corrected, adapted, and enhanced.

B.C.A study

Unit-2: Information Requirement Analysis

Information Requirements Analysis is the process of determining the data and information needs of an organization or system to support its goals and objectives. This involves identifying and defining the type, format, and content of the data and information required, as well as the sources of this information. The purpose of Information Requirements Analysis is to provide a comprehensive understanding of the information that is necessary to support the goals and objectives of a system, organization, or process, and to ensure that all necessary information is available, accurate, and accessible.

The process of Information Requirements Analysis typically includes the following steps:

1. **Identifying stakeholders:** This involves identifying the individuals and groups who will use or be impacted by the information and their specific requirements.
2. **Defining the business requirements:** This involves understanding the business goals and objectives and the information needed to support them.
3. **Determining the data sources:** This involves identifying the internal and external sources of data and information that will be used to support the business requirements.
4. **Evaluating the existing data:** This involves analyzing the data currently available and assessing its quality, accuracy, and relevance.
5. **Defining data and information requirements:** This involves specifying the type, format, and content of the data and information required to support the business requirements.
6. **Designing the information architecture:** This involves defining the structure, organization, and relationships of the data and information required to support the business requirements.
7. **Validation and verification:** This involves ensuring that the information requirements are accurate and complete and that the data sources and information architecture are suitable to support the business requirements.

Information Requirements Analysis is an important step in the development of any system, organization, or process as it ensures that the necessary information is available, accurate, and accessible to support the goals and objectives of the system.

Process modelling with physical logical data flow diagrams

Process modeling involves creating visual representations of business processes in order to understand and improve them. Physical and logical data flow diagrams are two types of diagrams used in process modeling.

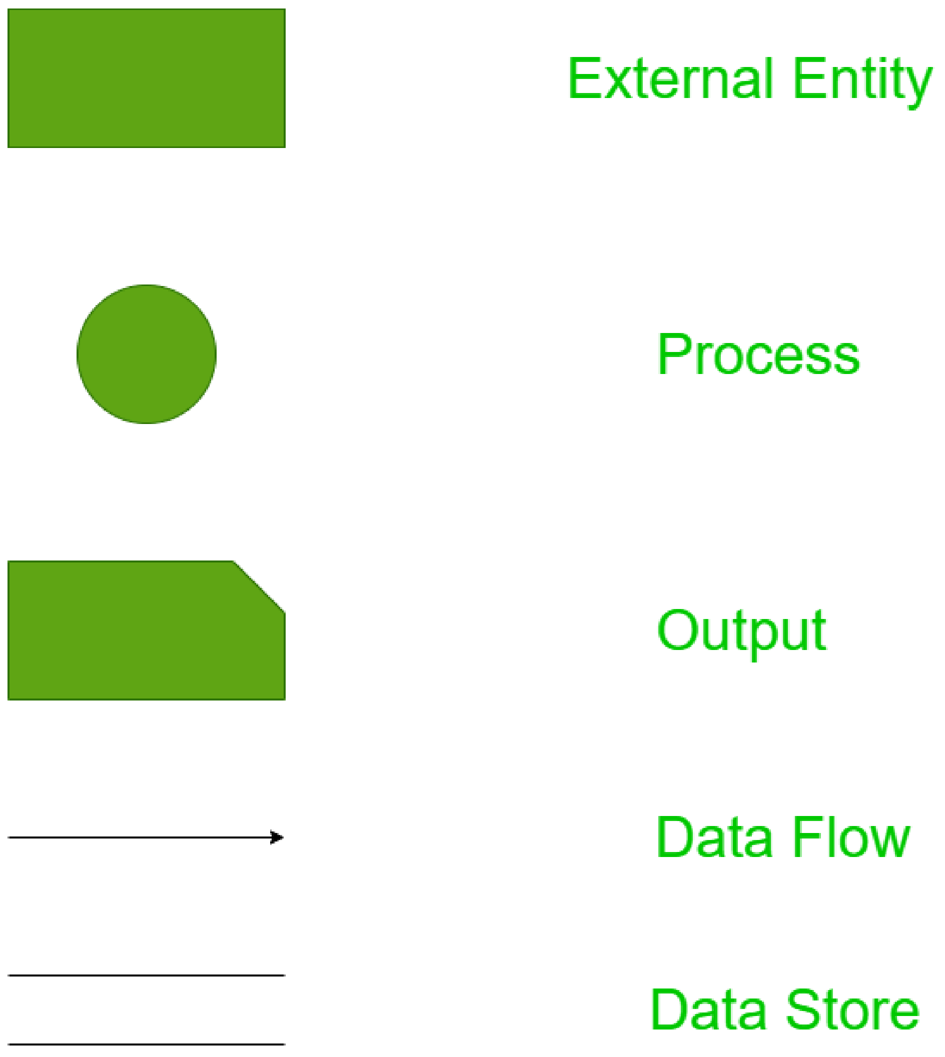
Physical Data Flow Diagrams (PDFDs) are graphical representations of the movement of data from its origin to its final destination. They depict the physical flow of information between people, systems, and storage media. PDFDs show how data is transformed from one format to another and how it is stored, retrieved, and transmitted.

Logical Data Flow Diagrams (LDPDs), on the other hand, depict the processing of data as it moves through a system. They show the logic behind the processing of data, including the transformations that take place, without emphasizing the physical aspect of the flow. LDFDs are used to identify the inputs, processing, and outputs of a system.

Both physical and logical data flow diagrams are useful in process modeling because they provide different perspectives on the flow of data in a system. PDFDs give a clear picture of the physical path that data takes, while LDFDs show the logic behind the processing of the data. By using both diagrams in conjunction, organizations can gain a complete understanding of their business processes and identify areas for improvement.

Components of Data Flow Diagram:

Following are the components of the data flow diagram that are used to represent source, destination, storage and flow of data.



- **Entities:**
  Entities include source and destination of the data. Entities are represented by rectangle with their corresponding names. **(https://practice.geeksforgeeks.org/courses/complete-interview-preparation?utm_source=geeksforgeeks&utm_medium=inarticle_cip_computersubjects&utm_campaign=inbound_promotions)**
- **Process:**
  The tasks performed on the data is known as process. Process is represented by circle. Somewhere round edge rectangles are also used to represent process.
- **Data Storage:**
  Data storage includes the database of the system. It is represented by rectangle with both smaller sides missing or in other words within two parallel lines.

- **Data Flow:**
  The movement of data in the system is known as data flow. It is represented with the help of arrow. The tail of the arrow is source and the head of the arrow is destination.

Importance of Data Flow Diagram:
Data flow diagram is a simple formalism to represent the flow of data in the system. It allows a simple set of intuitive concepts and rules. It is an elegant technique that is useful to represent the results of structured analysis of software problem as well as to represent the flow of documents in an organization.

Data modeling is the process of creating a conceptual representation of data and the relationships between data elements. Logical Entity Relationship Diagrams (ERDs) are a type of data model that is used to depict the relationships between entities in a system.

An entity is an object or concept that exists within the system and has attributes, or characteristics, that describe it. In a logical ERD, entities are represented by rectangles, and the relationships between them are represented by lines connecting the entities.

The relationships between entities in a logical ERD can be one-to-one, one-to-many, or many-to-many. These relationships are depicted using arrows on the lines connecting the entities. For example, a one-to-many relationship might be represented by an arrow pointing from a “Department” entity to an “Employee” entity, indicating that one department can have many employees, but each employee belongs to only one department.

Logical ERDs are useful for understanding the relationships between entities in a system, and for communicating this understanding to others. They can also be used to design and implement a database, as well as to validate the design and ensure that it meets the needs of the business.

Overall, logical ERDs are an important tool in data modeling, as they help organizations understand the relationships between data elements and design databases that are efficient and meet their business needs.

ER (Entity Relationship) Diagram in DBMS

- ER model stands for an Entity-Relationship model. It is a high-level data model. This model is used to define the data elements and relationship for a specified system.
- It develops a conceptual design for the database. It also develops a very simple and easy to design view of data.
- In ER modeling, the database structure is portrayed as a diagram called an entity-relationship diagram.

For example, Suppose we design a school database. In this database, the student will be an entity with attributes like address, name, id, age, etc. The address can be another entity with attributes like city, street name, pin code, etc and there will be a relationship between them.

DBMS ER model concept

Component of ER Diagram

DBMS ER model concept

1. Entity:

An entity may be any object, class, person or place. In the ER diagram, an entity can be represented as rectangles.

Consider an organization as an example- manager, product, employee, department etc. can be taken as an entity.

DBMS ER model concept

a. Weak Entity Play Video

(https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

https://imasdk.googleapis.com/js/core/bridge3.553.0_en.html#goog_988288928 (https://imasdk.googleapis.com/js/core/bridge3.553.0_en.html#goog_988288928)

An entity that depends on another entity called a weak entity. The weak entity doesn't contain any key attribute of its own. The weak entity is represented by a double rectangle.

DBMS ER model concept

2. Attribute

The attribute is used to describe the property of an entity. Eclipse is used to represent an attribute.

For example, id, age, contact number, name, etc. can be attributes of a student.

DBMS ER model concept

a. Key Attribute

The key attribute is used to represent the main characteristics of an entity. It represents a primary key. The key attribute is represented by an ellipse with the text underlined.

DBMS ER model concept

b. Composite Attribute

An attribute that composed of many other attributes is known as a composite attribute. The composite attribute is represented by an ellipse, and those ellipses are connected with an ellipse.

DBMS ER model concept

c. Multivalued Attribute

An attribute can have more than one value. These attributes are known as a multivalued attribute. The double oval is used to represent multivalued attribute.

For example, a student can have more than one phone number.

DBMS ER model concept

d. Derived Attribute

An attribute that can be derived from other attribute is known as a derived attribute. It can be represented by a dashed ellipse.

For example, A person's age changes over time and can be derived from another attribute like Date of birth.

DBMS ER model concept

3. Relationship

A relationship is used to describe the relation between entities. Diamond or rhombus is used to represent the relationship.

DBMS ER model concept

Types of relationship are as follows:

a. One-to-One Relationship

When only one instance of an entity is associated with the relationship, then it is known as one to one relationship.

For example, A female can marry to one male, and a male can marry to one female.

DBMS ER model concept

b. One-to-many relationship

When only one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then this is known as a one-to-many relationship.

For example, Scientist can invent many inventions, but the invention is done by the only specific scientist.

DBMS ER model concept

c. Many-to-one relationship

When more than one instance of the entity on the left, and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship.

For example, Student enrolls for only one course, but a course can have many students.

DBMS ER model concept

d. Many-to-many relationship

When more than one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship.

For example, Employee can assign by many projects and project can have many employees.

DBMS ER model concept

•••

B.C.A study

Unit-3: Designing software solutions

Refining the software Specification

Refining the software specification is the process of making improvements to the software specification document after the initial review has taken place. This step is important to ensure that the software meets all requirements and functions as intended. Refining the software specification may involve incorporating feedback from the review, making changes to the design to accommodate new requirements, or updating functional specifications based on the latest technology advancements.

During the refining process, the software specification document is re-evaluated and updated to ensure that it is accurate, complete, and consistent. The goal of refining the software specification is to create a clear and concise document that serves as a guide for the development of the software.

It is important to note that the refining process should continue throughout the software development lifecycle, as requirements and technology advancements may change over time. Regular updates to the software specification document ensure that the software development remains on track and that the end product meets the needs of the user.

In conclusion, refining the software specification is a critical step in the software development process and helps to ensure that the software is developed to meet the requirements and expectations of the user.

Application of fundamental design concept for data

The application of fundamental design concepts for data refers to the application of key design principles to the organization and representation of data in a software system. The following are some of the fundamental design concepts that are commonly applied to data:

1. **Abstraction:** Abstraction is the process of encapsulating data and operations into a single entity, such as an object or class, to hide its implementation details and simplify the interface.
2. **Encapsulation:** Encapsulation is the practice of bundling data and methods that manipulate the data within the same object or class, creating a self-contained unit of code.
3. **Modularity:** Modularity refers to the division of a software system into smaller, independent components that can be developed, tested, and maintained separately.
4. **Normalization**: Normalization is the process of organizing data in a relational database to eliminate redundancy and improve data integrity.
5. **Reusability:** Reusability is the ability to use existing components and modules in multiple applications, reducing development time and cost.

These fundamental design concepts are applied to data to improve the efficiency, scalability, and maintainability of a software system. By organizing data in a structured and consistent manner, software developers can ensure that the data can be easily accessed, processed, and updated, leading to a more efficient and effective software system.

In conclusion, the application of fundamental design concepts for data is a crucial step in the software development process and helps to ensure that data is organized and represented in a manner that supports the goals and objectives of the software system.

architectural and procedural designs using software blue print methodology and object oriented design paradigm

Architectural Design:

Architectural design is the process of defining the structure of a software system. It defines how the system components will interact with each other, how the data will flow between components, and what components will make up the system.

The software blueprint methodology uses a modular approach to architectural design. It breaks down the system into smaller, more manageable components that can be developed and tested independently. Each component has a well-defined interface that other components can use to communicate with it.

The object-oriented design paradigm focuses on the design of objects and their interactions. This is achieved by creating classes and objects, which are representations of real-world objects and their behaviors.

Procedural Design:

Procedural design is the process of defining the procedures, or algorithms, that the software system will use to perform its tasks. It defines the steps that the software will take to complete a task and the order in which those steps will be executed.

The software blueprint methodology uses a step-by-step approach to procedural design. It defines the procedures in detail, including input, output, and error handling. This helps to ensure that the procedures are clear and easy to understand, and that the system will behave consistently and predictably.

The object-oriented design paradigm focuses on encapsulating procedures within objects. This allows the procedures to be tied to the objects they manipulate, making it easier to understand how the procedures interact with the data.

In summary, the software blueprint methodology and object-oriented design paradigm work together to provide a structured, modular approach to software design. This helps to ensure that the software is well-organized, easy to understand, and easy to maintain.

Creating design document

A design document is a comprehensive document that outlines the design of a software system. It serves as a blueprint for the software development process, providing a clear and detailed description of the system's architecture, components, procedures, and interfaces.

The following is an example of a design document for a simple online shopping system:

1. **System Architecture:** This section describes the overall structure of the system, including the main components, their relationships, and the data flow between them. For example, the architecture may include a user interface, a database, and a server.
2. **Component Design:** This section provides detailed descriptions of each component, including its functions, inputs, outputs, and error handling. For example, the user interface component may be responsible for displaying product information and accepting user input.
3. **Procedures:** This section outlines the procedures that the system will use to complete tasks, such as adding products to the cart, processing payments, and sending confirmation emails.
4. **Interfaces:** This section describes the interfaces that the system components will use to communicate with each other, such as APIs or protocols.
5. **Data Model:** This section defines the data structures that the system will use to store and manipulate information, such as product information, user information, and order information.
6. **Error Handling:** This section outlines the strategies for handling errors and exceptions, such as logging errors and displaying error messages to the user.
7. **Testing:** This section describes the testing strategies and test cases that will be used to validate the system's behavior and ensure that it meets its requirements.

The design document should be written in a clear and concise manner, and should be reviewed by the software development team to ensure that it accurately reflects the intended design. It should also be updated as the design evolves and changes during the development process

Review of conformance to software requirements and quality.

A review of conformance to software requirements and quality is a process of evaluating a software system to ensure that it meets the specified requirements and standards for functionality, reliability, performance, and usability. The following are some key components of a software review process:

1. **Requirements review:** This involves verifying that the software meets the requirements specified in the software requirement specification (SRS) document. This includes checking for completeness, consistency, and correctness of the requirements.
2. **Design review:** This involves evaluating the software design to ensure that it is able to meet the requirements specified in the SRS document. This includes checking the architecture, interfaces, and data structures of the software.
3. **Code review**: This involves evaluating the source code to ensure that it is of high quality and adheres to programming best practices and coding standards. This includes checking for errors, bugs, and security vulnerabilities in the code.
4. **Test review:** This involves evaluating the testing processes and testing results to ensure that the software has been adequately tested and is of high quality. This includes checking for completeness and coverage of the test cases, and the accuracy of the test results.
5. **Usability review:** This involves evaluating the user interface and user experience of the software to ensure that it is easy to use and meets the needs of the end-users.
6. **Performance review**: This involves evaluating the performance of the software to ensure that it meets the specified performance requirements and standards. This includes checking for speed, scalability, and resource utilization.
7. **Reliability review:** This involves evaluating the reliability of the software to ensure that it can operate correctly and continuously under various conditions and loads.

B.C.A study

Unit-4 : Software implementetion

Software Implementation refers to the process of designing, coding, testing, and deploying software applications to a specific target environment. This process involves transforming the design and architecture of a software system into a functioning and fully operational software system.

The following are the key stages involved in software implementation:

1. **Requirements Gathering:** The first stage of software implementation is to gather the requirements of the software system. This includes understanding the business requirements, technical specifications, and user needs that the software system must meet.
2. **Design:** In this stage, the software architects and developers create a detailed design of the software system. The design includes the architecture, user interfaces, and the software components that make up the system.
3. **Coding:** In this stage, the software developers write the code for the software system based on the design. They use programming languages and tools to create the software components and ensure that the software system meets the requirements and design specifications.
4. **Testing:** Testing is a critical part of software implementation as it helps to identify any bugs or defects in the software system. The software is tested in various stages, such as unit testing, integration testing, system testing, and user acceptance testing.
5. **Deployment:** After the software has been thoroughly tested, it is deployed to the target environment, such as a client's data center, a cloud environment, or a production environment. The software is installed and configured in the target environment and made available for use.
6. **Maintenance:** Once the software is deployed, it must be maintained to ensure that it continues to function as expected. This involves fixing any bugs or defects, updating the software to meet changing requirements, and making changes to the software to improve performance or functionality.

In conclusion, software implementation is a complex and multi-stage process that involves several stages, from gathering requirements to deploying the software and maintaining it over time. It requires skilled software developers, software architects, and testers to ensure that the software system meets the business requirements and delivers the desired outcomes.

Relationship between design and implementation

The relationship between design and implementation is crucial in software development. Design is the blueprint of the software system, while implementation is the process of turning that design into a functioning software system.

Design provides the foundation for the software system, including its architecture, components, and user interfaces. It lays out the plan for how the software system will work and what it will look like. Implementation takes the design and creates a functional software system by writing code, testing it, and deploying it to a target environment.

Design and implementation are interdependent and must be closely linked to ensure that the software system meets the business requirements and delivers the desired outcomes. A well-designed software system that is not properly implemented will not be effective, and an poorly designed software system will be difficult to implement and will not deliver the desired outcomes.

Implementation issues and programming support environment

Implementation issues refer to the challenges and obstacles encountered during the deployment of a software system. Some of the common implementation issues are:

1. Inadequate Planning: The lack of proper planning and preparation can result in implementation delays and budget overruns.
2. **Compatibility Issues:** The software system may not be compatible with the existing hardware, software, or infrastructure, leading to implementation problems.
3. **Integration Issues:** The software system may not be able to integrate with other systems, causing compatibility problems.
4. **Technical Challenges:** Technical challenges such as data migration, data loss, and system downtime can arise during implementation.
5. **User Acceptance:** The end-users may not be accepting of the new software system, leading to resistance and decreased productivity.
6. **Cost Overruns:** The cost of the implementation may exceed the budget, resulting in financial strain on the organization.

A programming support environment refers to the tools, resources, and services available to support the development and deployment of software systems. Some of the key components of a programming support environment include:

1. **Development Tools:** This includes integrated development environments (IDEs), code editors, compilers, and debuggers.
2. **Testing Tools:** This includes software testing tools such as automated testing frameworks, code coverage tools, and load testing tools.
3. **Source Control:** Source control is a version control system that allows developers to manage and track changes to their code.
4. **Collaboration Tools:** Collaboration tools allow developers to work together on a software project, such as issue tracking systems, chat tools, and version control systems.
5. **Performance Optimization Tools:** This includes tools that help optimize the performance of the software system, such as profiling tools, memory analyzers, and performance tuning tools.
6. **Technical Support:** Technical support is essential to ensure that the software system is deployed smoothly and that any problems are quickly addressed.

Coding the procedural design

Coding the procedural design refers to the process of translating the high-level procedural design into a specific programming language. The following steps outline the process of coding the procedural design:

1. Choose a Programming Language: The first step in coding the procedural design is to choose a programming language that best suits the requirements of the project. This will depend on the type of software system, the hardware platform, and the development team's experience.
2. Review the Procedural Design: Before starting the coding process, it's essential to thoroughly review the procedural design to ensure that all the requirements are understood and that there are no missing components.
3. Write the Code: The next step is to write the code. This involves defining variables, writing functions and procedures, and creating data structures. It's important to follow best practices in coding, such as using meaningful variable names, commenting the code, and writing clear and concise code.
4. Test the Code: After the code has been written, it's essential to test it to ensure that it meets the requirements of the procedural design. This can be done using automated testing tools, manual testing, or a combination of both.
5. Debug the Code: If any issues are found during testing, they must be fixed. This process is known as debugging and involves identifying the source of the problem, fixing the code, and retesting to ensure that the issue has been resolved.
6. Document the Code: Documentation is a critical aspect of software development and helps to ensure that the code is understandable, maintainable, and reusable. This involves writing comments and documentation in the code and creating separate documentation for the software system.
7. Refine and Optimize: Once the code is working correctly, it can be refined and optimized. This involves making improvements to the code to make it more efficient, removing any unused code, and optimizing the performance of the software system.

Good coding style and review of correctness and readability.

Good coding style refers to the consistent and standard format and structure of the code, making it easier to read, understand and maintain. It covers aspects such as naming conventions, indentation, commenting, and organization of code.

Code Review of Correctness involves checking the code for any errors or bugs and ensuring that it meets the requirements and performs as intended. This includes checking for any potential security vulnerabilities, performance issues, and scalability concerns.

Code Review of Readability focuses on making sure that the code is easy to understand, follow and maintain. This includes using meaningful and descriptive variable names, breaking down complex logic into smaller, manageable chunks, and commenting code to provide context and explanation.

Having a good coding style and conducting regular code reviews helps improve the overall quality of the code, reduces the likelihood of bugs, and makes it easier for others to contribute and maintain the code in the future.

•••

B.C.A study

Unit-5:software maintenance

Software maintenance is widely accepted part of SDLC now a days. It stands for all the modifications and updations done after the delivery of software product. There are number of reasons, why modifications are required, some of them are briefly mentioned below:

- **Market Conditions** – Policies, which changes over the time, such as taxation and newly introduced constraints like, how to maintain bookkeeping, may trigger need for modification.
- **Client Requirements** – Over the time, customer may ask for new features or functions in the software.
- **Host Modifications** – If any of the hardware and/or platform (such as operating system) of the target host changes, software changes are needed to keep adaptability.
- **Organization Changes** – If there is any business level change at client end, such as reduction of organization strength, acquiring another company, organization venturing into new business, need to modify in the original software may arise.

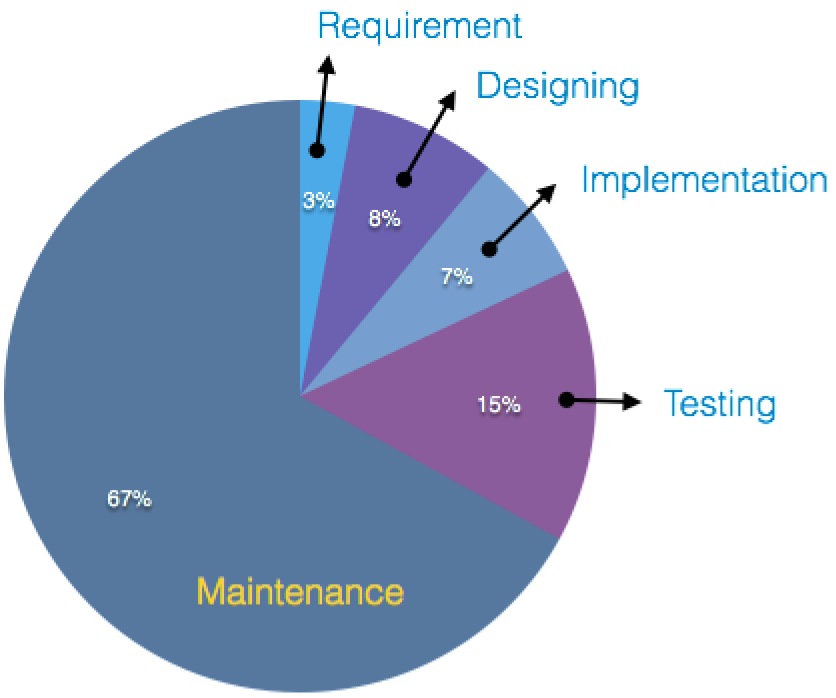
Types of maintenance

In a software lifetime, type of maintenance may vary based on its nature. It may be just a routine maintenance tasks as some bug discovered by some user or it may be a large event in itself based on maintenance size or nature. Following are some types of maintenance based on their characteristics:

- **Corrective Maintenance** – This includes modifications and updations done in order to correct or fix problems, which are either discovered by user or concluded by user error reports.
- **Adaptive Maintenance** – This includes modifications and updations applied to keep the software product up-to date and tuned to the ever changing world of technology and business environment.
- **Perfective Maintenance** – This includes modifications and updates done in order to keep the software usable over long period of time. It includes new features, new user requirements for refining the software and improve its reliability and performance.
- **Preventive Maintenance** – This includes modifications and updations to prevent future problems of the software. It aims to attend problems, which are not significant at this moment but may cause serious issues in future.

Cost of Maintenance

Reports suggest that the cost of maintenance is high. A study on estimating software maintenance found that the cost of maintenance is as high as 67% of the cost of entire software process cycle.



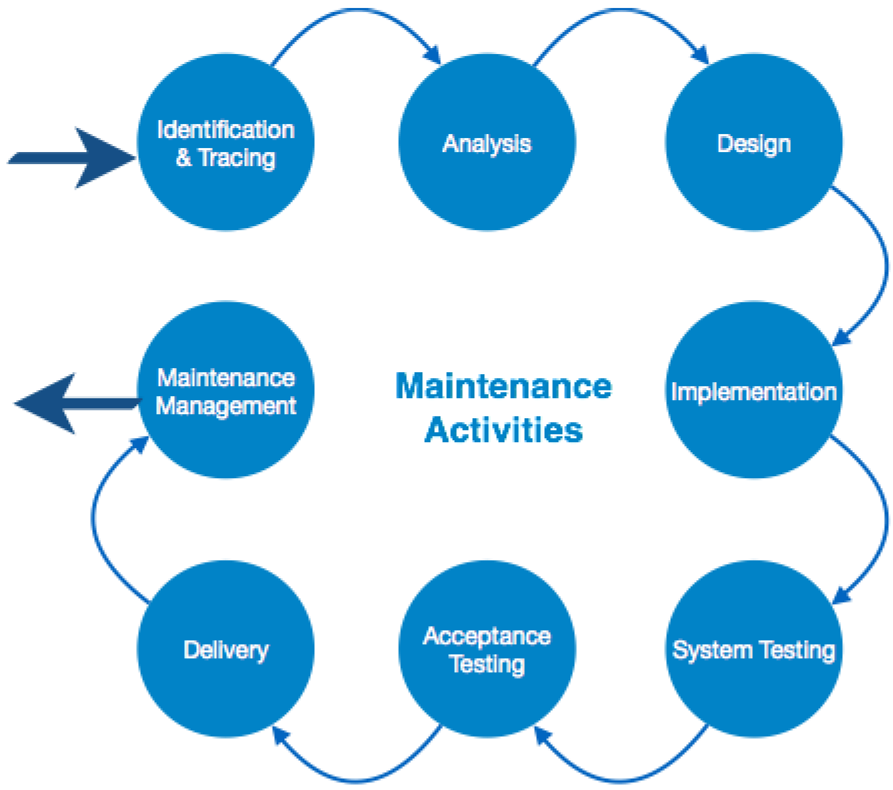
On an average, the cost of software maintenance is more than 50% of all SDLC phases. There are various factors, which trigger maintenance cost go high, such as:

Real-world factors affecting Maintenance Cost

- The standard age of any software is considered up to 10 to 15 years.
- Older softwares, which were meant to work on slow machines with less memory and storage capacity cannot keep themselves challenging against newly coming enhanced softwares on modern hardware.
- As technology advances, it becomes costly to maintain old software.
- Most maintenance engineers are newbie and use trial and error method to rectify problem.
- Often, changes made can easily hurt the original structure of the software, making it hard for any subsequent changes.
- Changes are often left undocumented which may cause more conflicts in future.

Designing Maintenance Activities

IEEE provides a framework for sequential maintenance process activities. It can be used in iterative manner and can be extended so that customized items and processes can be included.



These activities go hand-in-hand with each of the following phase:

- **Identification & Tracing** – It involves activities pertaining to identification of requirement of modification or maintenance. It is generated by user or system may itself report via logs or error messages.Here, the maintenance type is classified also.
- **Analysis** – The modification is analyzed for its impact on the system including safety and security implications. If probable impact is severe, alternative solution is looked for. A set of required modifications is then materialized into requirement specifications. The cost of modification/maintenance is analyzed and estimation is concluded.
- **Design** – New modules, which need to be replaced or modified, are designed against requirement specifications set in the previous stage. Test cases are created for validation and verification.
- **Implementation** – The new modules are coded with the help of structured design created in the design step.Every programmer is expected to do unit testing in parallel.
- **System Testing** – Integration testing is done among newly created modules. Integration testing is also carried out between new modules and the system. Finally the system is tested as a whole, following regressive testing procedures.

- **Acceptance Testing** – After testing the system internally, it is tested for acceptance with the help of users. If at this state, user complaints some issues they are addressed or noted to address in next iteration.
- **Delivery** – After acceptance test, the system is deployed all over the organization either by small update package or fresh installation of the system. The final testing takes place at client end after the software is delivered.Training facility is provided if required, in addition to the hard copy of user manual.
- **Maintenance management** – Configuration management is an essential part of system maintenance. It is aided with version control tools to control versions, semi-version or patch management.

Techniques for maintenance

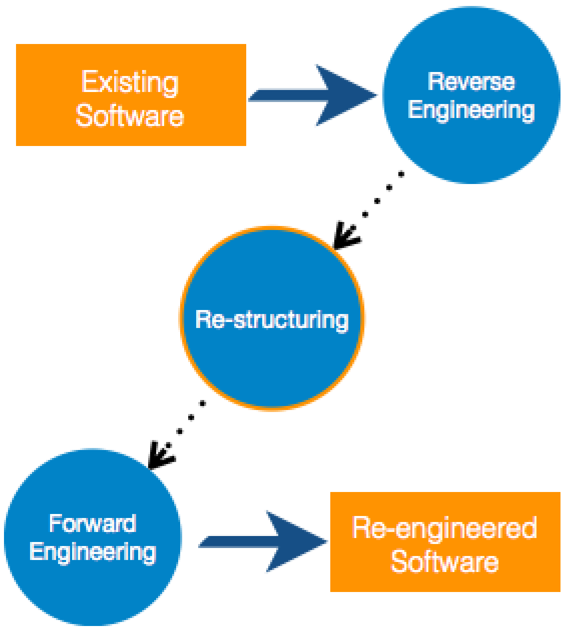
Software Re-engineering

When we need to update the software to keep it to the current market, without impacting its functionality, it is called software re-engineering. It is a thorough process where the design of software is changed and programs are re-written.

Legacy software cannot keep tuning with the latest technology available in the market. As the hardware become obsolete, updating of software becomes a headache. Even if software grows old with time, its functionality does not.

For example, initially Unix was developed in assembly language. When language C came into existence, Unix was re-engineered in C, because working in assembly language was difficult.

Other than this, sometimes programmers notice that few parts of software need more maintenance than others and they also need re-engineering.



Re-Engineering Process

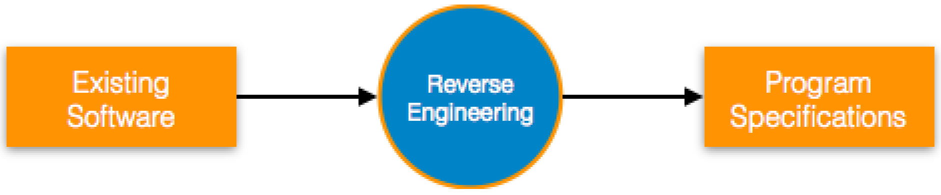
- **Decide** what to re-engineer. Is it whole software or a part of it?
- **Perform** Reverse Engineering, in order to obtain specifications of existing software.
- **Restructure Program** if required. For example, changing function-oriented programs into object-oriented programs.
- **Re-structure data** as required.
- **Apply Forward engineering** concepts in order to get re-engineered software.

There are few important terms used in Software re-engineering

Reverse Engineering

It is a process to achieve system specification by thoroughly analyzing, understanding the existing system. This process can be seen as reverse SDLC model, i.e. we try to get higher abstraction level by analyzing lower abstraction levels.

An existing system is previously implemented design, about which we know nothing. Designers then do reverse engineering by looking at the code and try to get the design. With design in hand, they try to conclude the specifications. Thus, going in reverse from code to system specification.



Program Restructuring

It is a process to re-structure and re-construct the existing software. It is all about re-arranging the source code, either in same programming language or from one programming language to a different one. Restructuring can have either source code-restructuring and data-restructuring or both.

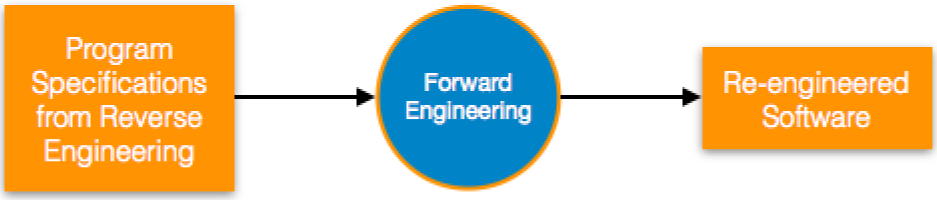
Re-structuring does not impact the functionality of the software but enhance reliability and maintainability. Program components, which cause errors very frequently can be changed, or updated with re-structuring.

The dependability of software on obsolete hardware platform can be removed via re-structuring.

Forward Engineering

Forward engineering is a process of obtaining desired software from the specifications in hand which were brought down by means of reverse engineering. It assumes that there was some software engineering already done in the past.

Forward engineering is same as software engineering process with only one difference – it is carried out always after reverse engineering.



Component reusability

A component is a part of software program code, which executes an independent task in the system. It can be a small module or sub-system itself.

Example

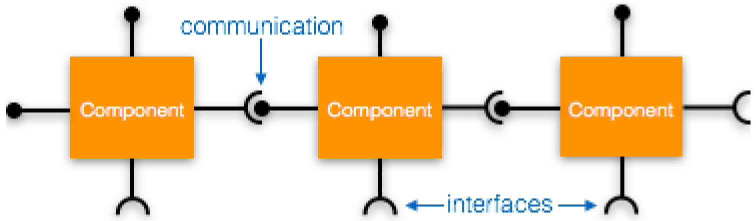
The login procedures used on the web can be considered as components, printing system in software can be seen as a component of the software.

Components have high cohesion of functionality and lower rate of coupling, i.e. they work independently and can perform tasks without depending on other modules.

In OOP, the objects are designed are very specific to their concern and have fewer chances to be used in some other software.

In modular programming, the modules are coded to perform specific tasks which can be used across number of other software programs.

There is a whole new vertical, which is based on re-use of software component, and is known as Component Based Software Engineering (CBSE).

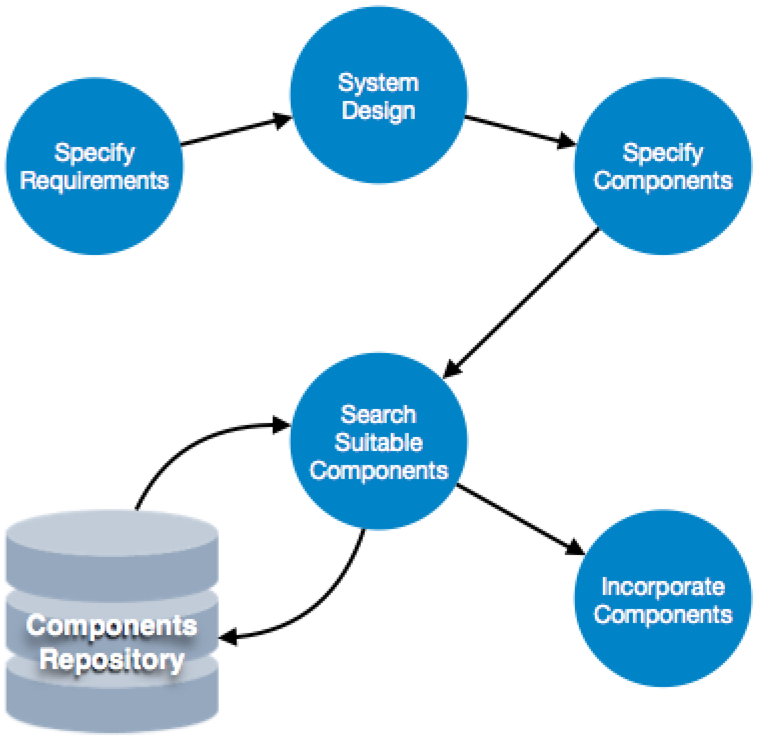


Re-use can be done at various levels

- **Application level** – Where an entire application is used as sub-system of new software.
- **Component level** – Where sub-system of an application is used.
- **Modules level** – Where functional modules are re-used.Software components provide interfaces, which can be used to establish communication among different components.

Reuse Process

Two kinds of method can be adopted: either by keeping requirements same and adjusting components or by keeping components same and modifying requirements.



- **Requirement Specification** – The functional and non-functional requirements are specified, which a software product must comply to, with the help of existing system, user input or both.
- **Design** – This is also a standard SDLC process step, where requirements are defined in terms of software parlance. Basic architecture of system as a whole and its sub-systems are created.
- **Specify Components** – By studying the software design, the designers segregate the entire system into smaller components or sub-systems. One complete software design turns into a collection of a

huge set of components working together.

- **Search Suitable Components** – The software component repository is referred by designers to search for the matching component, on the basis of functionality and intended software requirements..
- **Incorporate Components** – All matched components are packed together to shape them as complete software.

•••