CAVITE STATE UNIVERSITY

CAVITE CITY CAMPUS

Learning Module in

DCIT26 - APPLICATION DEVELOPMENT AND EMERGING TECHNOLOGIES

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE OUTLINE

COURSE OUTLINE

CHAPTER 1

INTRODUCTION TO SOFTWARE DEVELOPMENT

OBJECTIVES

At the end of the chapter, the student will be able to:

1. *identify different types of software applications;*
2. *understand the evolution of software and hardware; and*
3. *interpret the different software application type*

I. WHAT IS A COMPUTER SOFTWARE?

Computer software – refers to a program that enables a computer to perform a specific task, as opposed to the physical components of the system (hardware). This includes application software such as a word processor, which enables a user to perform a task, and system software such as an operating system, which enables other software to run properly, by interfacing with hardware and with other software.

History of Computer Software

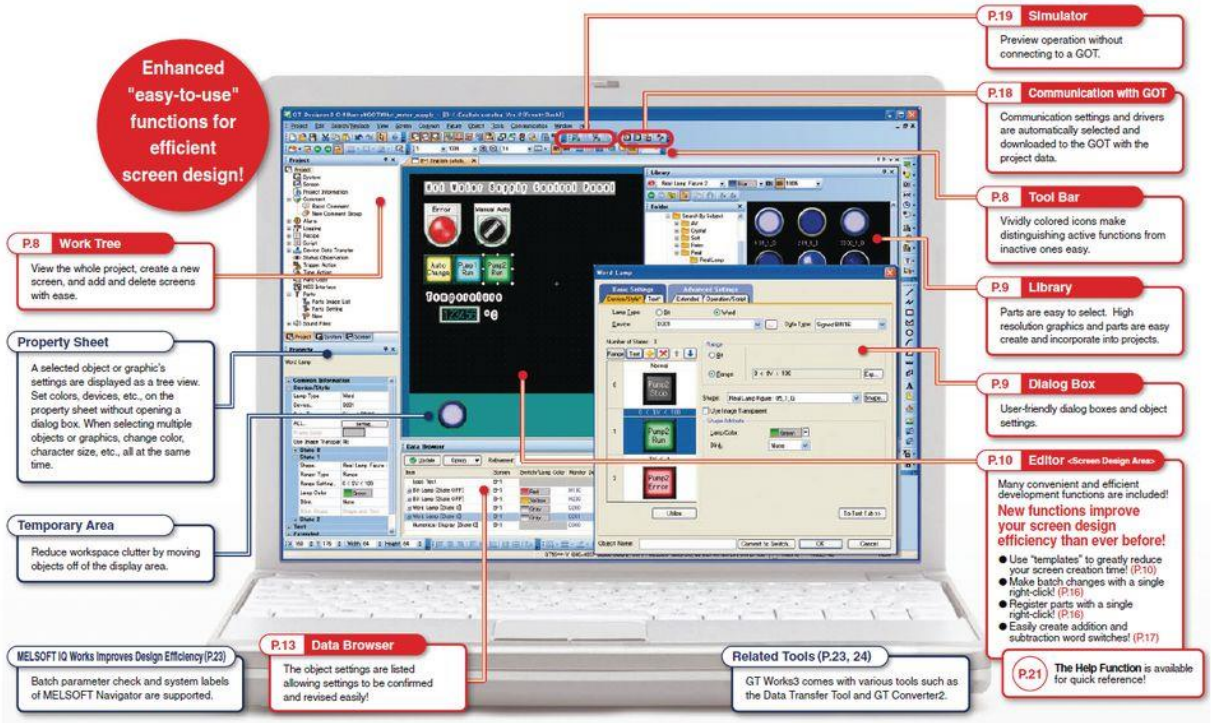
The origins of software architecture as a concept were first identified in the research work of Edsger Dijkstra in 1968, and David Parnas in the early 1970s. The scientists emphasized that the structure of a software system matters and getting the structure right is critical. The study of the field increased in popularity since the early 1990s, with research work concentrating on architectural styles (patterns), architecture description languages, architecture documentation, and formal methods.

Mary Shaw and David Garlan of Carnegie Mellon wrote a book titled, Software Architecture: Perspectives on an Emerging Discipline in 1996, which brought forward the concepts in Software Architecture, such as components, connectors, styles, and so on. UCI's Institute for Software Research's efforts in software architecture research is directed primarily in architectural styles, architecture description languages, and dynamic architectures.

Types of Computer Software:

a. ***System software*** helps run the computer hardware and computer system. It includes operating systems, device drivers, diagnostic tools, servers, windowing systems, utilities, and more. The purpose of systems software is to insulate the applications programmer as much as possible from the details of the particular computer complex being used, especially memory and other hardware features, and such accessory devices as communications, printers, readers, displays, keyboards, etc.

b. ***Programming software*** usually provides tools to assist a programmer in writing computer programs and software using different programming languages in a more convenient way. The tools include text editors, compilers, interpreters, linkers, debuggers, and so on. An Integrated development environment (IDE) merges those tools into a software bundle, and a programmer may not need to type multiple commands for compiling, interpreter, debugging, tracing, and etc., because the IDE usually has an advanced graphical user interface (GUI).



c. ***Application software*** allows humans to accomplish one or more specific (non-computer related) tasks. Typical applications include industrial automation, business software, educational software, medical software, databases, and computer games. Businesses are probably the biggest users of application software, but almost every field of human activity now uses some form of application software. It is used to automate all sorts of functions.

Computer Software Processes and Method

1. **Software Specifications**

   In this process, detailed description of a software system to be developed with its functional and non-functional requirements.

2. **Software Development**

   In this process, designing, programming, documenting, testing, and bug fixing is done.

3. **Software Validation**

   In this process, evaluation software product is done to ensure that the software meets the business requirements as well as the end user's needs.

4. **Software Evolution**

   It is a process of developing software initially, then timely updating it for various reasons.

Application software - refers to a type of computer program that performs a specific personal, educational, and business function. Each program is designed to assist the user with a particular process, which may be related to productivity, creativity, and/or communication.

II. WHAT IS OBJECT-ORIENTED ANALYSIS AND DESIGN?

Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.

The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions. They are modelled after real-world objects that the system interacts with. In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

Grady Booch has defined OOA as, "Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain".

The primary tasks in object-oriented analysis (OOA) are:

- Identifying objects
- Organizing the objects by creating object model diagram
- Defining the internals of the objects, or object attributes
- Defining the behavior of the objects, i.e., object actions
- Describing how the objects interact

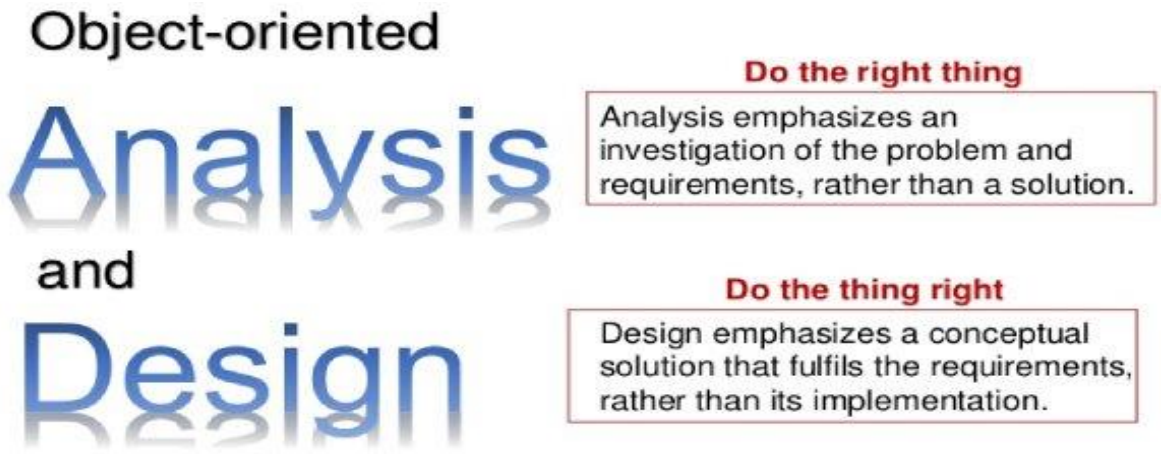
The common models used in OOA are use cases and object models.

Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology−independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.

The implementation details generally include:

- Restructuring the class data (if necessary),
- Implementation of methods, i.e., internal data structures and algorithms,
- Implementation of control, and
- Implementation of associations.

Grady Booch has defined object-oriented design as “a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design”.



Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object–oriented programming is:

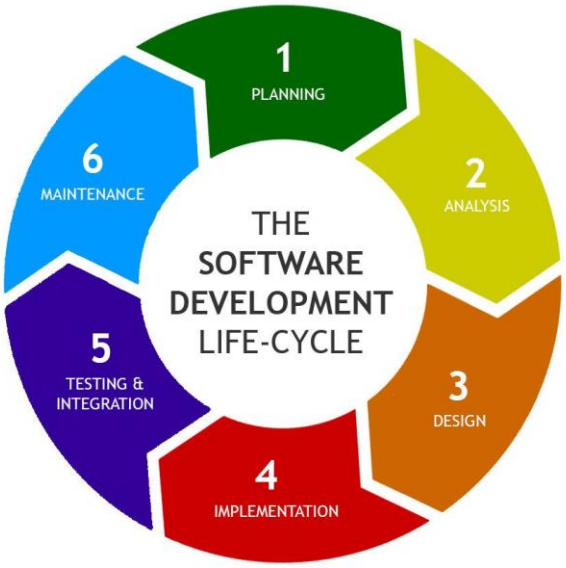
- Bottom–up approach in program design
- Programs organized around objects, grouped in classes
- Focus on data with methods to operate upon object’s data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes

III. Software development life cycle

SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software.

The life cycle defines a methodology for improving the quality of software and the overall development process.

The following figure is a graphical representation of the various stages of a typical SDLC.



Stage 1: Planning and Requirement Analysis

Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational and technical areas.

Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage. The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

Stage 2: Defining Requirements

Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through an SRS (Software Requirement Specification) document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the Product Architecture

SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than

one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity, budget and time constraints, the best design approach is selected for the product.

A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third-party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS.

Stage 4: Building or Developing the Product

In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.

Developers must follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers, etc. are used to generate the code. Different high-level programming languages such as C, C++, Pascal, Java and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Stage 5: Testing the Product

This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However, this stage refers to the testing only stage of the product where product defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance

Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometimes product deployment happens in stages as per the business strategy of that organization. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).

Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

SDLC Models

There are various software development life cycle models defined and designed which are followed during the software development process. These models are also referred as Software Development Process Models". Each process model follows a Series of steps unique to its type to ensure success in the process of software development.

Following are the most important and popular SDLC models followed in the industry −

- Waterfall Model
- Iterative Model
- Spiral Model
- V-Model
- Big Bang Model

Other related methodologies are Agile Model, RAD Model, Rapid Application Development and Prototyping Models.

IV. Software engineering myths and misconception

Many causes of a software affliction can be traced to a mythology that arose during the early history of software development. Unlike ancient myths that often provide human lessons well worth heeding, software myths propagated misinformation and confusion. Software myths had a number of attributes that made them insidious; for instance, they appeared to be reasonable statements of fact (sometimes containing elements of truth), they had an intuitive feel, and they were often promulgated by experienced practitioners who "knew the score."

Today, most knowledgeable professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and technical people alike. However, old attitudes and habits are difficult to modify, and remnants of software myths are still believed.

Management myths. Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

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

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

Myth: My people have state-of-the-art software development tools, after all, we buy them the newest computers.

Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.

Myth: If we get behind schedule, we can add more programmers and catch up (sometimes called the Mongolian horde concept).

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [BRO75]: "adding people to a late software project makes it “In the absence of meaningful standards, a new industry like software comes to depend instead on folklore.” Tom DeMarco later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

Myth: If I decide to outsource3 the software project to a third party, I can just relax and let that firm build it.

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

Customer myths. A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

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

Reality: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces,

design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

Myth: Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. If serious attention is given to up-front definition, early requests for change can be accommodated easily. The customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly. Resources have been committed and a design framework has been established. Change can cause upheaval that requires additional resources and major design modification, that is, additional cost. Changes in function, performance, interface, or other characteristics during implementation (code and test) have a severe impact on cost. Change, when requested after software is in production, can be over an order of magnitude more expensive than the same change requested earlier.

Practitioner's myths. Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software, programming was viewed as an art form. Old ways and attitudes die hard.

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

Reality: Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done." Industry data ([LIE80], [JON91], [PUT97]) indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

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

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

Myth: The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a software configuration that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

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

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

CHAPTER **2**

REQUIREMENTS ANALYSIS AND MODELING

OBJECTIVES

At the end of the chapter, the student will be able to:

1. *compare the different software engineering activities;*
2. *use different system models in given software applications; and*
3. *classify the software requirements engineering and understand the need of each requirement*

I. REQUIREMENTS MODELING: GUIDING PRINCIPLES

Requirements play a fundamental role in the life cycle of systems. In particular, the various development disciplines (such as architecture, design, implementation, and testing) are based mainly on the requirements of the system as specified during requirements engineering and are largely dependent on the quality of these requirements.

According to the IREB Glossary of Requirements Engineering Terminology [Glin2011], a requirement is (1) a need that is perceived by a stakeholder or (2) a capability or property that a system must have. Requirements engineering is concerned with ensuring that the requirements of the system under development are formulated as completely, correctly, and precisely as possible, thereby providing optimal support for the other development disciplines and activities in the life cycle of the system.

There are three basic principles of requirement modelling:

1. Viewpoints

This principle states to understand the viewpoints of the stakeholders before specifying the requirements. We need to understand what are the actual problems of stakeholders or their business.

2. Analysis

This principle states to convert gathered raw data into contract style documented lists of requirements and eventually visualize the system before even it has been constructed.

3. Feasibility

This principle states to validate if the requirements fulfill the organizational objectives. It also checks whether the current system will be transformed into the intended or targeted system if the gathered requirements are implemented.

II. OVERVIEW AND ELEMENTS

What is Requirements Engineering?

The software requirements are description of features and functionalities of the target system. Requirements convey the expectations of users from the software product. The requirements can be obvious or hidden, known or unknown, expected or unexpected from client's point of view.

Requirement Engineering is the process of gathering the software requirements from client, analyze and document them is known as requirement engineering. The goal of requirement engineering is to develop and maintain sophisticated and descriptive 'System Requirements Specification' document.

Requirement Engineering Process

It is a four-step process, which includes

- Feasibility Study
- Requirement Gathering
- Software Requirement Specification
- Software Requirement Validation

Feasibility study

When the client approaches the organization for getting the desired product developed, it comes up with rough idea about what all functions the software must perform and which all features are expected from the software.

Referencing to this information, the analysts do a detailed study about whether the desired system and its functionality are feasible to develop.

This feasibility study is focused towards goal of the organization. This study analyzes whether the software product can be practically materialized in terms of implementation, contribution of project to organization, cost constraints and as per values and objectives of the organization. It explores technical aspects of the project and product such as usability, maintainability, productivity and integration ability.

The output of this phase should be a feasibility study report that should contain adequate comments and recommendations for management about whether or not the project should be undertaken.

Requirement Gathering

If the feasibility report is positive towards undertaking the project, next phase starts with gathering requirements from the user. Analysts and engineers communicate with the client and end-users to know their ideas on what the software should provide and which features they want the software to include.

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

The Role of Collaboration

Team collaboration accelerates product development and allows you to get the product to market faster, since employees address difficulties together.

Team collaboration drives creative thinking and effective brainstorming. It allows you to look at problems from multiple angles and points of view.

Collaborative teams hold regular meetings, so they have a shared and clear vision of the direction they're going.

Every party involved knows the full scope of the project and what they're accountable for. This allows each team member to perform efficiently and saves time, money, and effort.

Use Cases

A use case is a written description of how users will perform tasks on your website. It outlines, from a user's point of view, a system's behavior as it responds to a request. Each use case is represented as a sequence of simple steps, beginning with a user's goal and ending when that goal is fulfilled.

Benefits of Use Cases

Use cases add value because they help explain how the system should behave and, in the process, they also help brainstorm what could go wrong. They provide a list of goals and this list can be used to establish the cost and complexity of the system. Project teams can then negotiate which functions become requirements and are built.

What Use Cases Include	What Use Cases Do NOT Include
• Who is using the website • What the user want to do • The user's goal • The steps the user takes to accomplish a particular task • How the website should respond to an action	• Implementation-specific language • Details about the user interfaces or screens.

Elements of a Use Case

Depending on how in depth and complex you want or need to get, use cases describe a combination of the following elements:

Actor – anyone or anything that performs a behavior (who is using the system)

Stakeholder – someone or something with vested interests in the behavior of the system under discussion (SUD)

Primary Actor – stakeholder who initiates an interaction with the system to achieve a goal

Preconditions – what must be true or happen before and after the use case runs.

Triggers – this is the event that causes the use case to be initiated.

Main success scenarios [Basic Flow] – use case in which nothing goes wrong.

Alternative paths [Alternative Flow] – these paths are a variation on the main theme. These exceptions are what happen when things go wrong at the system level.

How to Write a Use Case

Write the steps in a use case in an easy-to-understand narrative. Kenworthy (1997) outlines the following steps:

1. Identify who is going to be using the website.
2. Pick one of those users.
3. Define what that user wants to do on the site. Each thing the use does on the site becomes a use case.
4. For each use case, decide on the normal course of events when that user is using the site.
5. Describe the basic course in the description for the use case. Describe it in terms of what the user does and what the system does in response that the user should be aware of.
6. When the basic course is described, consider alternate courses of events and add those to "extend" the use case.
7. Look for commonalities among the use cases. Extract these and note them as common course use cases.
8. Repeat the steps 2 through 7 for all other users.

III. UNIFIED MODELING LANGUAGE MODELS

Model

Models are forms of description often adopted in software development. They are abstractions used to represent and communicate what is important, devoid of unnecessary detail, and to help developers deal with the complexity of the problem being investigated or the solution being developed.

The Unified Modeling Language (UML)

Unified Modeling Language, is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.

The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing object-oriented software and the software development process.

The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

Design of a UML Model

Structure diagrams show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other. The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts, there are seven types of structure diagram as follows:

- Class Diagram
- Component Diagram
- Deployment Diagram
- Object Diagram
- Package Diagram
- Composite Structure Diagram
- Profile Diagram

Behavior diagrams show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time, there are seven types of behavior diagrams as follows:

- Use Case Diagram
- Activity Diagram
- State Machine Diagram
- Sequence Diagram
- Communication Diagram

- Interaction Overview Diagram
- Timing Diagram

IV. DATA MODELING IN SOFTWARE ENGINEERING

Data Model vs Requirements Modeling

Requirements modeling in software engineering is part of analysis and design. It's the planning stage of developing a software application. Requirements modeling focuses on the 'what', not the 'how'. In other words, requirements modeling identifies what requirements the application must meet in order to be successful. Requirements modeling includes many sub-stages, one of them being data modeling. Generally speaking, requirements modeling will begin with scenario-based modeling, which results in creating a use case. A use case, simply put, is a primary example of how the software application will be used and what it is expected to do. Once a use case exists, one of the stages that follow will be data modeling. The **Data Model** is defined as an abstract model that organizes data description, data semantics, and consistency constraints of data. The data model emphasizes on what data is needed and how it should be organized instead of what operations will be performed on data. Data Model is like an architect's building plan, which helps to build conceptual models and set a relationship between data items.

Data Objects, Attributes, and Relationships

Entity

An **entity** is a distinguishable real-world ‘object’ that exists. An object should not be considered as an ‘entity’ until it can be easily identified from all other objects of the real world. In a database, only that ‘thing’ or ‘object’ is considered as an entity about which data can be stored or retrieved.

The entities sharing the same set of properties or same set of attributes are kept in one **entity set** which is also known as a relation or a table in the relational database. An entity in an entity set is represented by the sequence of attributes values which is simply a sequence of values hence, it is also called as a tuple.

Each entity sets are represented by a table. So, each row of a table in the database represents an entity. It is essential that a table must not contain duplicate entities as it will lead to ambiguity. Each entity in a table must be uniquely identified which means two entities in a relation, should not contain the same set of values for the same set of attributes.

An entity is further classified into two types, i.e. tangible entity and intangible entity:

1. A tangible entity is one which physically exists in real-world. Such as a person, student, bank locker, etc.

2. An intangible entity is one which exists logically in real-world such as bank account, reservation, email account etc.

The figure below shows the entities in the student table:

Roll No.	Name	Course	
CS08	Steive	Comp. Sci.	→ Entity 1
EE54	Jhoson	Electronics	→ Entity 2
B12	Eva	Biology	→ Entity 3
F32	Jhoson	Finance	→ Entity 4
M26	Erica	Maths	→ Entity 5

Student Table

Kinds of Entities

Independent entities, also referred to as kernels, are the backbone of the database. They are what other tables are based on. Kernels have the following characteristics:

- They are the building blocks of a database.
- The primary key may be simple or composite.
- The primary key is not a foreign key.
- They do not depend on another entity for their existence.

Dependent entities, also referred to as derived entities, depend on other tables for their meaning. These entities have the following characteristics:

- Dependent entities are used to connect two kernels together.
- They are said to be existence dependent on two or more tables.
- Many to many relationships become associative tables with at least two foreign keys.
- They may contain other attributes.
- The foreign key identifies each associated table.

Characteristic entities provide more information about another table. These entities have the following characteristics:

- They represent multivalued attributes.
- They describe other entities.
- They typically have a one to many relationships.
- The foreign key is used to further identify the characterized table.

Attributes

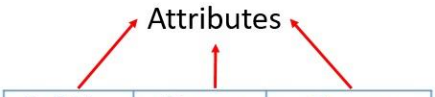
Attributes describe the characteristics or properties of an entity in a database table. An entity in a database table is defined with the 'fixed' set of attributes. For example, if we have to define a student entity then we can define it with the set of attributes like roll number, name, course. The attribute values, of each student entity, will define its characteristics in the table.

In a relational database, we store data in the form of tables. The column header of the table represents the attributes. A table must not have duplicate attributes. Each attribute in a table

has a certain domain which allows it to accept a certain 'set of values' only. In an entity, each attribute is allowed to have only one value which could be a number, text, date, time etc.

Attributes have further refinement such as keys. A single attribute or a set of attributes that can distinguish an entity in an entity set/relation is termed as key. In the database, we have a primary key, super key, composite key, foreign key.

The figure below it shows the attributes of the student table:

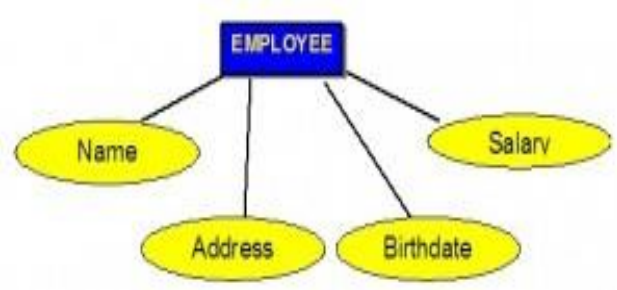


Roll No.	Name	Course
CS08	Steive	Comp. Sci.
EE54	Jhoson	Electronics
B12	Eva	Biology
F32	Jhoson	Finance
M26	Erica	Maths

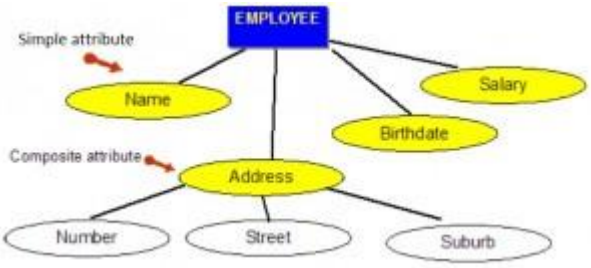
Student Table

Types of Attributes

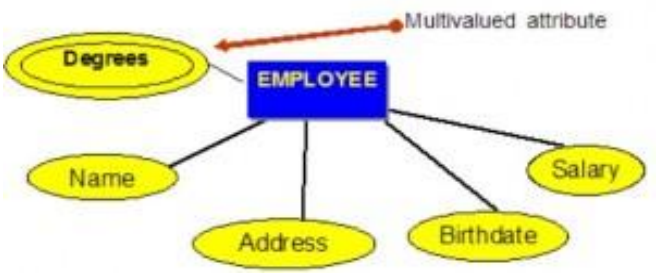
1. **Single Key attribute or primary attribute:** is an ID, key, letter or number that uniquely identifies that item. For example, it can be the number of a certain invoice (e.g. the individual ID of that invoice). A table that contains a single key attribute is considered a strong entity. However, a table might contain more than one key attribute if it's derived from other tables.



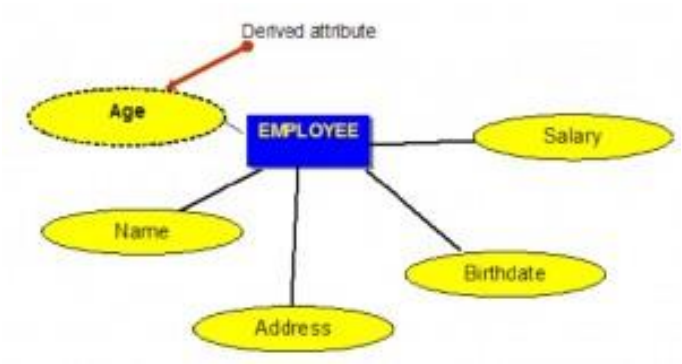
2. **Composite attribute:** is an attribute composed of several other simple attributes. For example, the Address attribute of an Employee entity could consist of the Street, City, Postal code and Country attributes.



3. **Multivalued attribute:** is an attribute where more than one description can be provided. For example, an Employee entity may have more than one Email ID attributes in the same cell.



4. **Derived attribute:** as the name implies, these are derived from other attributes, either directly or through specific formula results. For example, the Age attribute of an Employee could be derived from the Date of Birth attribute. In other instances, a formula might calculate the VAT of a certain payment, so that whenever the cell with the attribute Payment is filled, the cell with the derived attribute VAT automatically calculates its value.



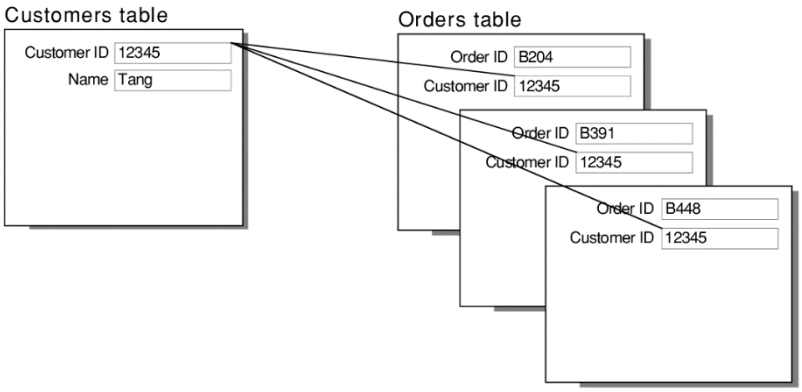
Relationships

Relationships are the glue that holds the tables together. They are used to connect related information between tables. A relationship is established between two database tables when one table uses a freign key that references the primary key of another table.

Types of Relationships

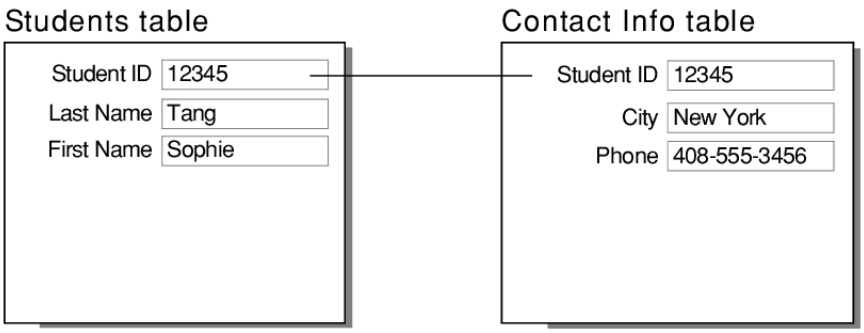
One to many (1:M) relationship

In a one-to-many relationship, one record in a table can be associated with one or more records in another table. For example, each customer can have many sales orders.



One to one (1:1) relationship

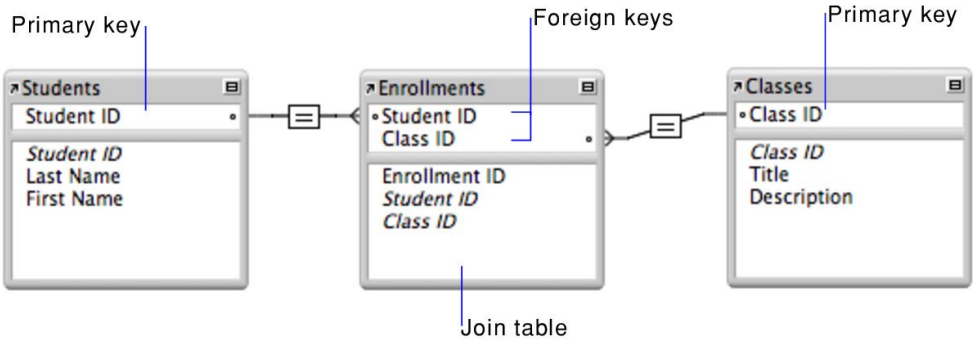
In a one-to-one relationship, one record in a table is associated with one and only one record in another table. For example, in a school database, each student has only one student ID, and each student ID is assigned to only one person.



Many to many (M:N) relationships

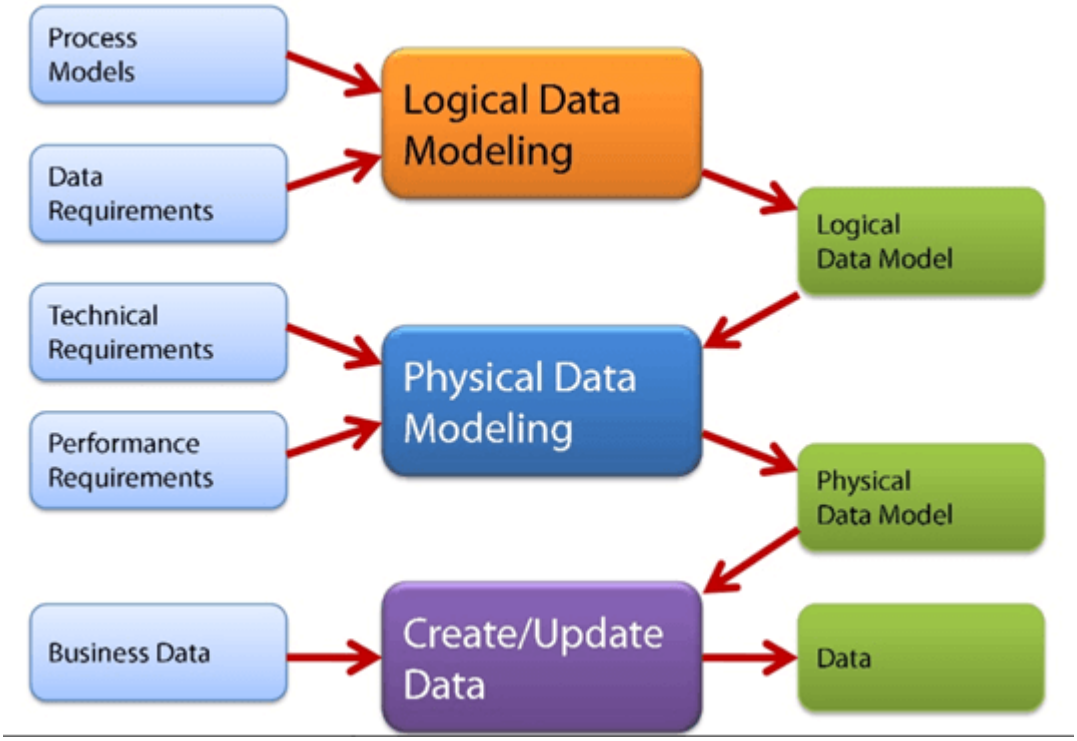
A many-to-many relationship occurs when multiple records in a table are associated with multiple records in another table. A typical example of a many-to many relationship is one between students and classes. A student can register for many classes, and a class can include many students.

The following example includes a Students table, which contains a record for each student, and a Classes table, which contains a record for each class. A join table, Enrollments, creates two one-to-many relationships—one between each of the two tables.



Conceptual, Logical, and Physical Modeling

1. **Conceptual Data Model:** This Data Model defines WHAT the system contains. This model is typically created by Business stakeholders and Data Architects. The purpose is to organize, scope and define business concepts and rules.
2. **Logical Data Model:** Defines HOW the system should be implemented regardless of the DBMS. This model is typically created by Data Architects and Business Analysts. The purpose is to developed technical map of rules and data structures.
3. **Physical Data Model:** This Data Model describes HOW the system will be implemented using a specific DBMS system. This model is typically created by DBA and developers. The purpose is actual implementation of the database.



Types of Data Model

Conceptual Data Model

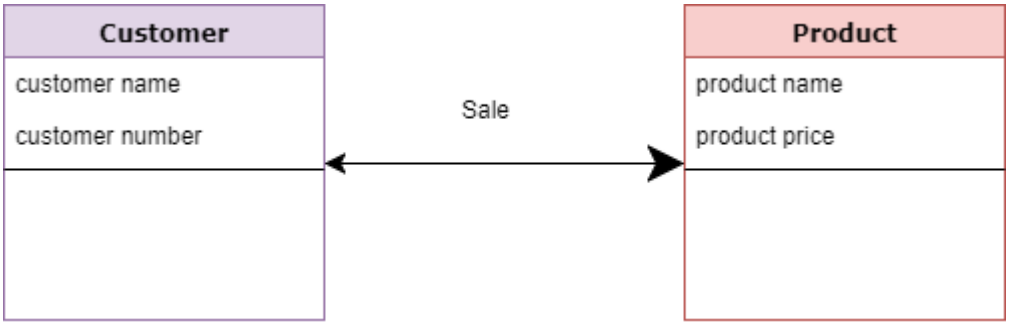
A **Conceptual Data Model** is an organized view of database concepts and their relationships. The purpose of creating a conceptual data model is to establish entities, their attributes, and relationships. In this data modeling level, there is hardly any detail available on the actual database structure. Business stakeholders and data architects typically create a conceptual data model.

The 3 basic tenants of Conceptual Data Model are

1. Entity: A real-world thing
2. Attribute: Characteristics or properties of an entity
3. Relationship: Dependency or association between two entities

Data model example:

- Customer and Product are two entities. Customer number and name are attributes of the Customer entity
- Product name and price are attributes of product entity
- Sale is the relationship between the customer and product



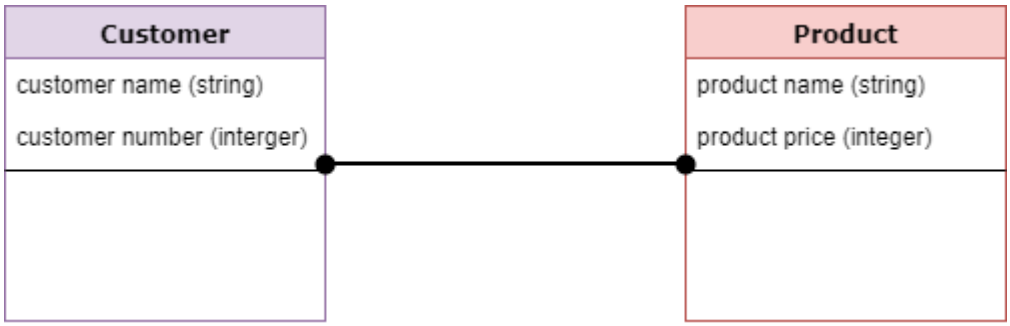
Conceptual Data Model

Characteristics of a conceptual data model

- Offers organization-wide coverage of the business concepts.
- This type of Data Model is designed and developed for a business audience.
- The conceptual model is developed independently of hardware specifications like data storage capacity, location or software specifications like DBMS vendor and technology. The focus is to represent data as a user will see it in the "real world."

Logical Data Model

The **Logical Data Model** is used to define the structure of data elements and to set relationships between them. The logical data model adds further information to the conceptual data model elements. The advantage of using a Logical data model is to provide a foundation to form the base for the Physical model. However, the modeling structure remains generic.



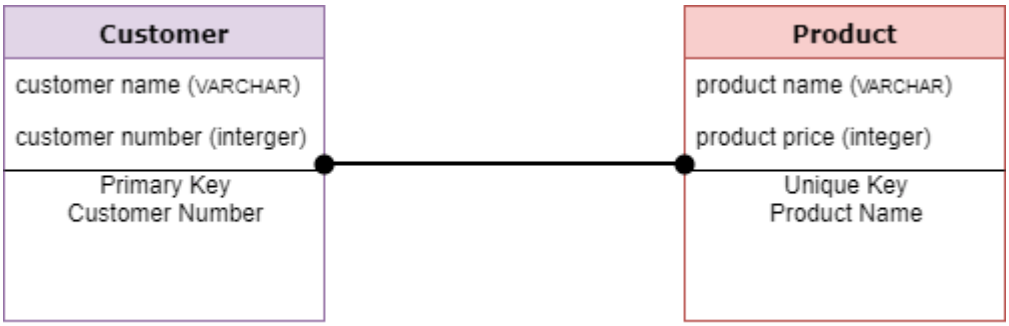
Logical Data Model

Characteristics of a Logical data model

- Describes data needs for a single project but could integrate with other logical data models based on the scope of the project.
- Designed and developed independently from the DBMS.
- Data attributes will have datatypes with exact precisions and length.

Physical Data Model

A **Physical Data Model** describes a database-specific implementation of the data model. It offers database abstraction and helps generate the schema. This is because of the richness of meta-data offered by a Physical Data Model. The physical data model also helps in visualizing database structure by replicating database column keys, constraints, indexes, triggers, and other RDBMS features.



Physical Data Model

Characteristics of a physical data model:

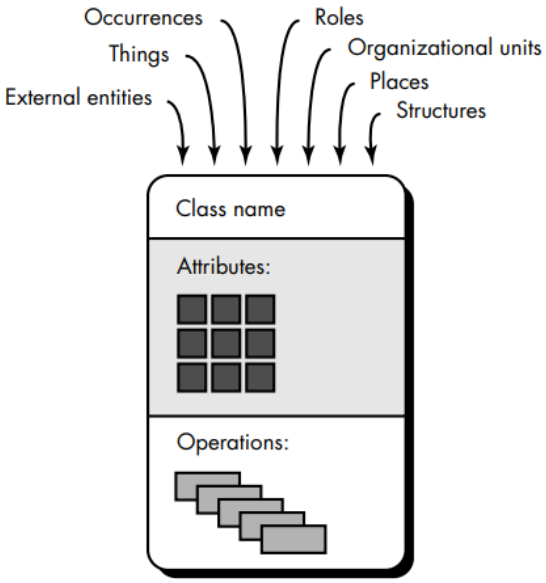
- The physical data model describes data need for a single project or application though it may be integrated with other physical data models based on project scope.
- Data Model contains relationships between tables that which addresses cardinality and nullability of the relationships.
- Developed for a specific version of a DBMS, location, data storage or technology to be used in the project.
- Columns should have exact datatypes, lengths assigned and default values.
- Primary and Foreign keys, views, indexes, access profiles, and authorizations, etc. are defined.

V. CLASS-BASED DATA MODELING

Class-based modeling is a stage of requirements modeling. In the context of software engineering, requirements modeling examines the requirements a proposed software application or system must meet in order to be successful. Typically, requirements modeling begins with scenario-based modeling, which develops a use case that will help with the next stages, like data and class-based modeling. Class-based modeling takes the use case and extracts from it the classes, attributes, and operations the application will use. Like all modeling stages, the end result of class-based modeling is most often a diagram or series of diagrams, most frequently created using UML, Unified Modeling Language.

Classes

A class is a description of a set of objects having similar attributes, operations, relationship and behavior. How objects manifest themselves.



Identifying Analysis Classes

1. Perform a grammatical parse of the problem statement or use cases.
2. Classes are determined by underlining each noun or noun clause.
3. A class required to implement a solution is a part of the solution space.
4. A class necessary only to describe a solution is part of the problem space.
5. A class should not have an imperative procedural name.
6. List the potential class names in a table and classify each according to some taxonomy and class selection characteristics
7. A potential class should satisfy nearly all of the selection characteristics to be considered a legitimate problem domain class.

General classifications to help identify elements that can be considered potential classes:

- **External entities** that produce or consume information to be used by the application.
- **Things** that are part of the information domain of the problem.
- **Occurrences** or events that occur within the context of system operation.
- **Roles** played by people who interact with the system.

- **Organizational units** that are relevant to an application. (Divisions, groups or teams, like the Sales and Editorial departments at Acme Publishing.)
- **Places** that establish the context of the problem and the overall function of the system.

1. **Structures** that define a class of objects or related classes of objects.

Selection Characteristics

Once the potential classes have been identified, six selection characteristics are used to decide which classes to include in the model:

1. Retained information - Information must be remembered about the system over time.
2. Needed services - A set of operations that can change the attributes of a class.
3. Multiple attributes - A class must have multiple attributes. If something identified as a potential class has only a single attribute, it is best included as a variable.
4. Common attributes - All instances of a class share the same attributes.
5. Common operations - A set of operations apply to all instances of a class.
6. Essential requirements - Entities that produce or consume information.

Specifying Attributes

- Attributes are the set of data object that fully define the class within the context of the problem.
- Attributes describes a class that has been selected for inclusion in the requirements model.

Defining Operations

Operations define the behavior of an object. It focuses on problem-oriented behavior rather than behaviors required for implementation.

CHAPTER **3**

DESIGN PRINCIPLES AND PATTERNS

OBJECTIVES

At the end of the chapter, the student will be able to:

1. *apply different types of software design;*
2. *execute the software reuse in given software application; and*
3. *construct a software design using object-oriented tools*

I. SOFTWARE DESIGN AND REUSE

Software reuse is the use of previously acquired concepts or objects in a new situation, it involves encoding development information at different levels of abstraction, storing this representation for future reference, matching of new and old situations, duplication of already developed objects and actions, and their adaptation to suit new requirements;

Reusability is a measure of the ease with which one can use those previous concepts or objects in the new situations.

The object of reusability, reusable artefact, can be any information which a developer may need in the process of creating software (Freeman 1983), this includes any of the following software components:

- code fragments, which come in a form of source code, PDL, or various charts;
- logical program structures, such as modules, interfaces, or data structures;
- functional structures, e.g. specifications of functions and their collections;
- domain knowledge, i.e. scientific laws, models of knowledge domains;
- knowledge of development process, in a form of life-cycle models;
- environment-level information, e.g. experiential data or users' feedback;
- artefact transformation during development process (Basili 1990); etc.

Advantage of Software Reuse

- Increase software productivity
- Shorten software development time
- Improve software system interoperability
- Develop software with fewer people
- Move personnel more easily from project to project
- Reduce software development and maintenance costs
- Produce more standardized software
- Produce better quality software and provide a powerful competitive advantage

What is design reuse?

In information technology, design reuse is the inclusion of previously designed components (blocks of logic or data) in software and hardware. The term is more frequently used in

hardware development. Design reuse makes it faster and cheaper to design and build a new product, since the reused components will not only be already designed but also tested for reliability. Developers can reuse a component in both similar and completely different applications.

Advantage of Design Reuse

- Increased Reliability
- Reduced process risk
- Effective use of Specialist
- Standard compliance
- Accelerated development
- Lower cost

II. Design process software engineering

From Art to Engineering

In the early days of software development, building software was more of an individual artistic process. This explains why one of the most famous and most widespread books among software developers was The Art of Computer Programming by Donald Knuth. While explaining in detail the computer programming process at that time, the title of the book indicates that programming was a creative artistic activity. However, with software projects becoming bigger and more complex, they had to be approached differently and more engineering techniques were employed.

Software is a predominant product in our lives that carries out highly critical tasks in our everyday life. Take, for example, the management of an airport or a hospital or a healthcare IoT monitoring system. All these are critical tasks that need engineering methodologies in order to be built and implemented.

Design in Software Engineering

In all engineering domains, design is a very important step that precedes building or implementing the product. For example, consider constructing a building. It is unimaginable that builders go straight to the field and start the construction before detailed designs are established by engineers. In fact, constructing a building without designing it beforehand would be dangerous and the building may have serious issues that could put people's lives in danger.

In software engineering, design is one phase of the software development methodology. In fact, it can be claimed that it is the most important phase of the whole process. This is because the entire product will be built upon decisions made in this phase.

Software design is the first step of the software development process. It can be defined as a high-level, technology independent abstraction which describes a system that will be able to accomplish the tasks that were identified in the requirement analysis phase. During this phase, designers will need to make compromises between suggested solutions and available resources. For example, they may have to adopt less efficient solutions due to lack of material or financial resources. Decisions related to the following aspects are made during the design process:

1. ***System Architecture***

   The system architecture defines the relationship between the different pieces (or modules) of the system. Usually, this is presented as a diagram that depicts the relationships and dependencies between the system components. These components are obtained from breaking down the abstract solution (that came from the requirement analysis) into realizable pieces.

2. ***Effort Estimation***

   In this step, designers will estimate the amount of work required to finish the work on the design problem. Usually, this is expressed using 'person/hour' which allows the designers to map the required effort with the deadlines as well as the financial resources.

3. ***Design Workflows***

   Here designers will use the dependency diagram from the system architecture to decide the order of the different steps to be taken for the development and deployment of the system.

4. ***Tools and Platforms***

   In the design phase, designers should make decisions on the tools and platforms to be used to develop and use the software product. For example, they can decide to use MVC.NET to develop a web application or use LAMP (Linux, Apache, MySQL, PHP).

5. ***Interfaces***

   User interfaces should also be designed. These are the screens that the end user will interact with.

6. ***Iterative Nature of the Design Process***

   There are different methodologies of software development. Figure 1 shows one of the most commonly adopted methodologies known as the Waterfall model.

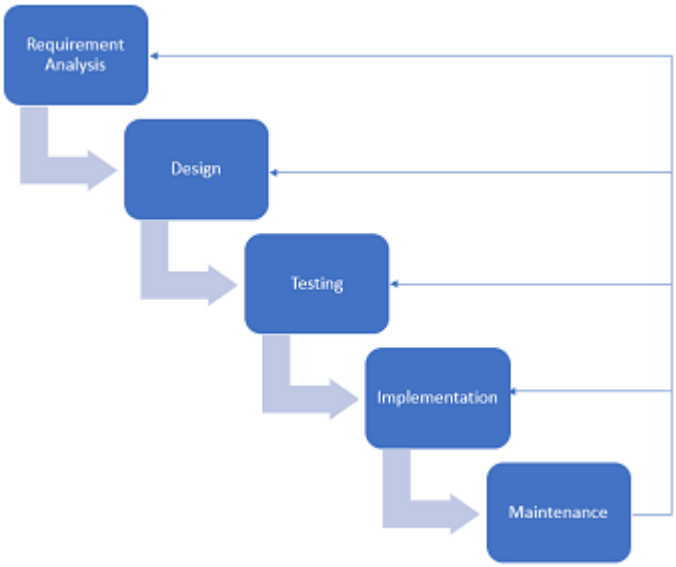


Figure 1: The Waterfall software development approach

In today's complicated software development environments, it is claimed that the Waterfall model needs to be adapted to the quickly changing market and client requirements. Thus, increasingly, more software development companies are adopting what is called the spiral or incremental development model, shown in Figure 2.

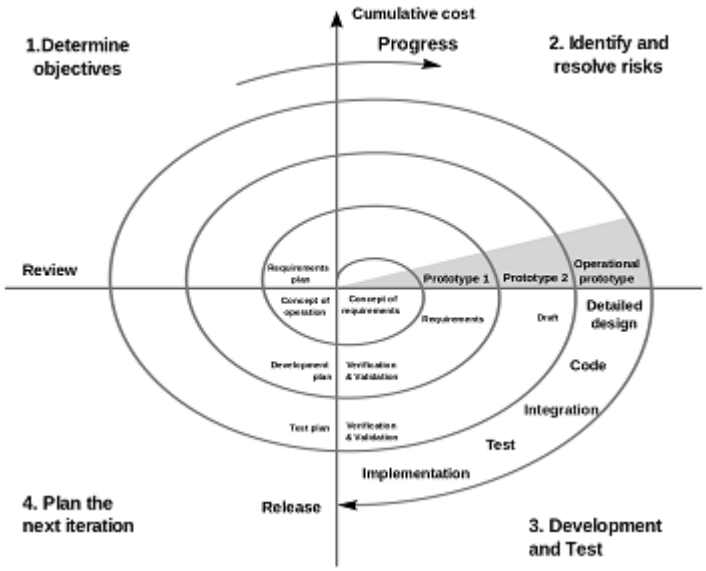


Figure 2: Incremental software development approach

III. DESIGN CONCEPTS

The Basic of Software Design

In the field of software development, there are many stages of planning and analysis before the project is finalized and development can formally begin. Design always comes before development, and functional design makes coding and maintenance very simple.

There are seven main principles to keep in mind in the design model in object-oriented programming (OOP):

- Abstraction
- Patterns
- Separation of data
- Modularity
- Data hiding
- Functional independence
- Refactoring

Each of these is an essential part of the design model and must be met if one wishes to develop a successful software system. Each principle must be considered and thoroughly reviewed before the testing phase of the software can begin.

Abstraction & Patterns

Abstraction, is the process of hiding complex properties or characteristics from the software itself to keep things more simplistic. This allows for a much higher level of efficiency for complex software designs since it allows the developers to list out only the necessary elements or objects required. In this principle, the developer will define the properties, type of functions, and the interface for each of said objects in the project. The developers will be able to hide the complicated and unnecessary details in the background while retaining core information in the foreground.

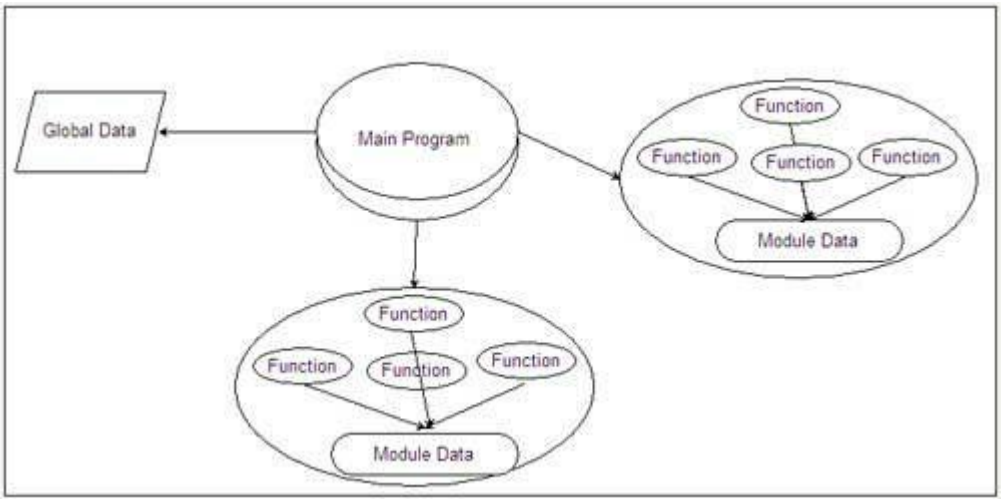
We use patterns to identify solutions to design problems that are recurring and can be solved reliably. A pattern must be guaranteed to work so that it may be reused many times over, but it also must be relevant to the current project at the same time. If the said pattern does not fit into the overall design function of the current project, it might be possible to reuse it as a guide to help create a new pattern that would be more fitting to the situation.

There are three main patterns:

- **Architectural**, which is a high-level pattern type that can be defined as the overall formation and organization of the software system itself.
- **Design**, which is a medium-level pattern type that is used by the developers to solve problems in the design stage of development. It can also affect how objects or components interact with one another.
- And, finally, **idioms**, which are low-level pattern types, often known as coding patterns, and are used as a workaround means of setting up and defining how components will be interacting with the software itself without being dependent on the programming language. There are many different programming languages all with different syntax rules, making this a requirement to function on a variety of platforms.

Modularity

Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as **modules.** A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules. After developing the modules, they are integrated together to meet the software requirements. Note that larger the number of modules a system is divided into, greater will be the effort required to integrate the modules.



Modularizing a design helps to plan the development in a more effective manner, accommodate changes easily, conduct testing and debugging effectively and efficiently, and conduct maintenance work without adversely affecting the functioning of the software.

DEVELOPING A DESIGN MODEL

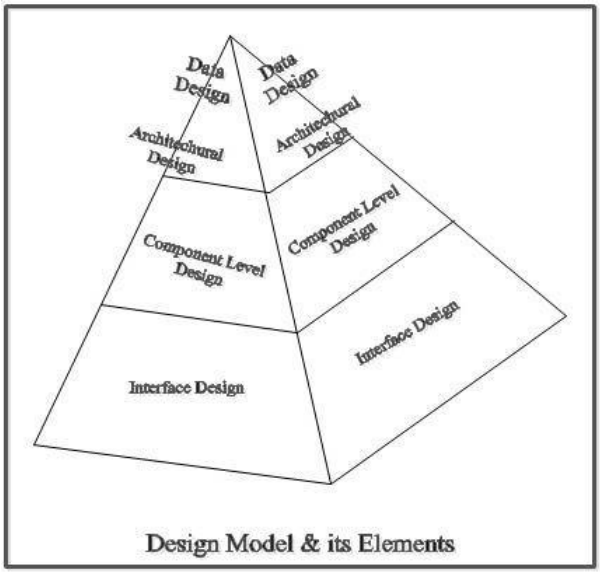
To develop a complete specification of design (design model), four design models are needed. These models are listed below.

- **Data design:** This specifies the data structures for implementing the software by converting data objects and their relationships identified during the analysis phase. Various studies suggest that design engineering should begin with data design, since this design lays the foundation for all other design models.

- **Architectural design:** This specifies the relationship between the structural elements of the software, design patterns, architectural styles, and the factors affecting the ways in which architecture can be implemented.

- **Component-level design:** This provides the detailed description of how structural elements of software will actually be implemented.

- **Interface design:** This depicts how the software communicates with the system that interoperates with it and with the end-users.



Design Model & its Elements

IV. Five basic concepts of object-oriented design

- **Object/Class:** A tight coupling or association of data structures with the methods or functions that act on the data. This is called a *class*, or *object* (an object is created based on a class). Each object serves a separate function. It is defined by its properties, what

it is and what it can do. An object can be part of a class, which is a set of objects that are similar.

- **Information hiding:** The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as *private* or *protected* to the owning *class*.
- **Inheritance:** The ability for a *class* to extend or override functionality of another *class*. The so-called *subclass* has a whole section that is derived (inherited) from the *superclass* and then it has its own set of functions and data.
- **Interface**): The ability to defer the implementation of a *method*. The ability to define the *functions* or *methods* signatures without implementing them.
- **Polymorphism**: The ability to replace an *object* with its *sub objects*. The ability of an *object-variable* to contain, not only that *object*, but also all of its *sub objects*.

V. SOFTWARE ARCHITECTURAL DESIGN

The software needs the architectural design to represents the design of software. Architectural design is defined 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:

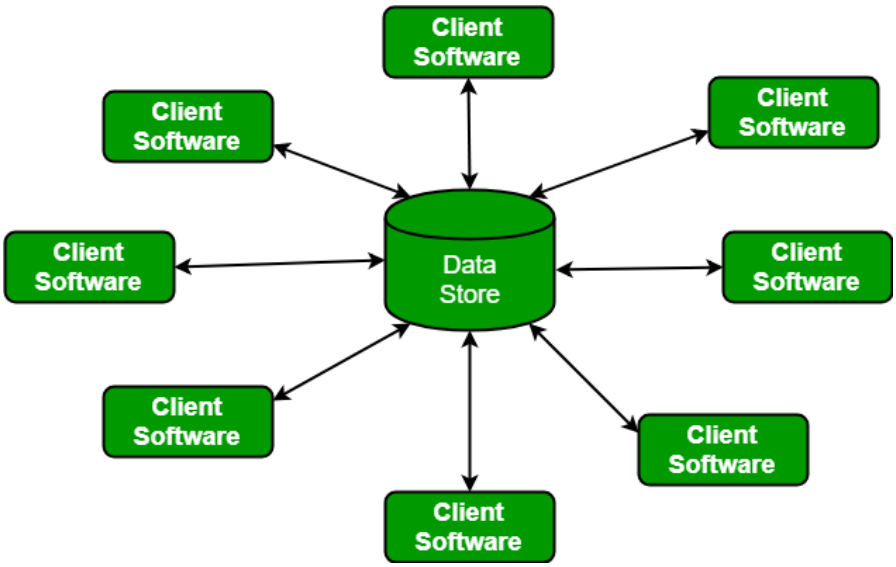
- A set of components (e.g.: a database, computational modules) that will perform a function required by the system.
- The set of connectors will help in coordination, communication, and cooperation between the components.
- Conditions that how components can be integrated to form the system.
- Semantic 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.

Taxonomy of Architectural styles:

1. Data centered architectures:

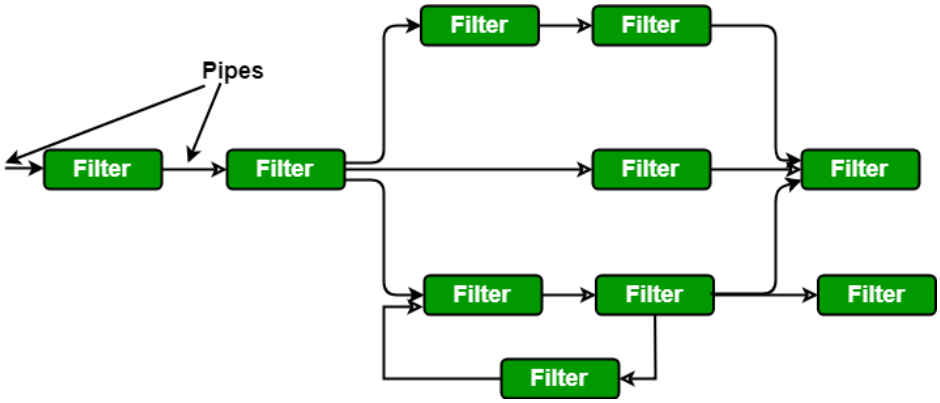
- A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store.
- The figure illustrates a typical data centered style. The client software accesses a central repository. Variation of this approach are used to transform the repository into a blackboard when data related to client or data of interest for the client change the notifications to client software.
- This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.
- Data can be passed among clients using blackboard mechanism.



2. Data flow architectures:

- This kind of architecture is used when input data to be transformed into output data through a series of computational manipulative components.
- The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by pipes.
- Pipes are used to transmit data from one component to the next.

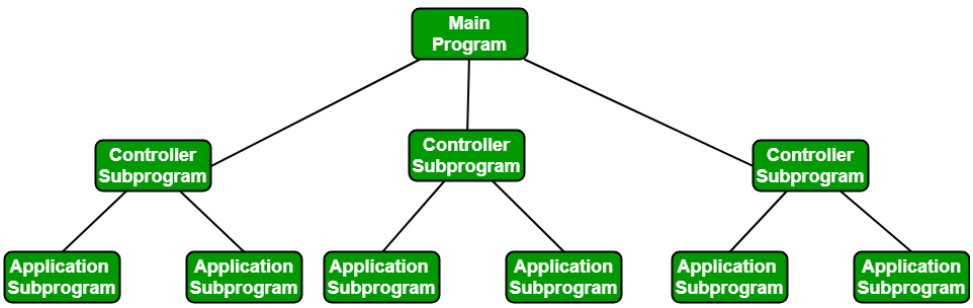
- Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighboring filters.
- If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.



3. **Call and Return architectures:**

It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.

- **Remote procedure call architecture:** These components is used to present in a main program or sub program architecture distributed among multiple computers on a network.
- **Main program or Subprogram architectures:** The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.

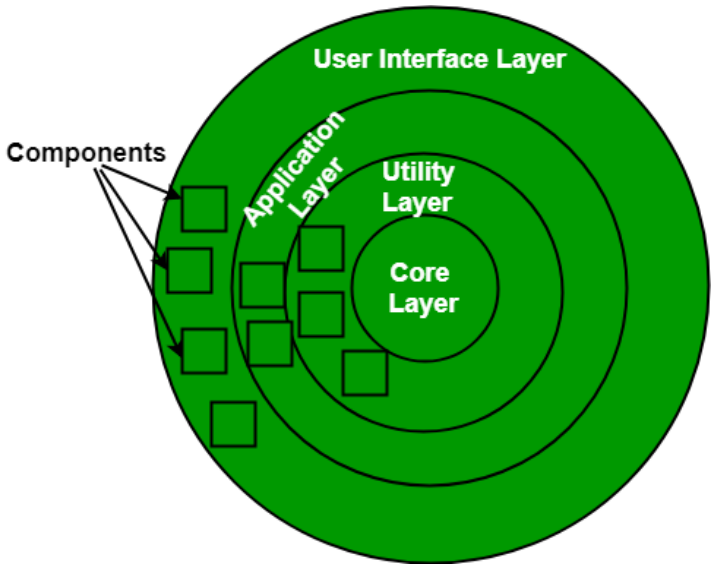


4. **Object Oriented architecture:**

The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.

5. **Layered architecture:**

- A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.
- At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing (communication and coordination with OS)
- Intermediate layers to utility services and application software functions.



CHAPTER 4

PROTOTYPING AND QUALITY ASSURANCES

OBJECTIVES

At the end of the chapter, the student will be able to:

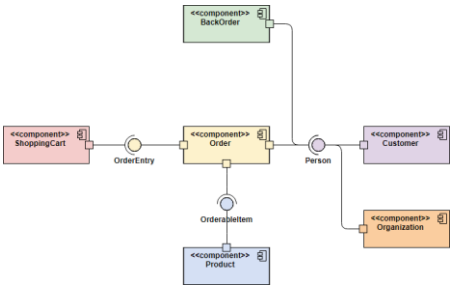
1. *create a component-level design;*
2. *develop an application using user interface design tools; and*
3. *create designs using different types of prototyping*

I. COMPONENT-LEVEL DESIGN SOFTWARE ENGINEERING

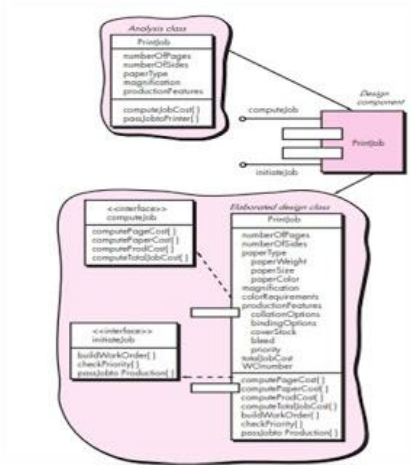
Component-based architecture focuses on the decomposition of the design into individual functional or logical components that represent well-defined communication interfaces containing methods, events, and properties. It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions.

The primary objective of component-based architecture is to ensure **component reusability**. A component encapsulates functionality and behaviors of a software element into a reusable and self-deployable binary unit.

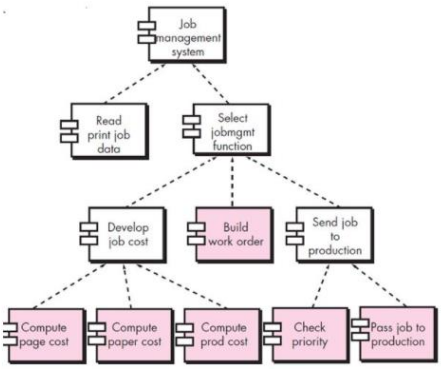
- Component is a modular (module), deployable, replaceable part of a system that encapsulates implementation and exposes a set of interfaces.



- Object-oriented view is that component contains a set of collaborating classes
    1. Each elaborated class includes all attributes and operations relevant to its implementation.
    2. All interfaces communication and collaboration with other design classes are also defined.
    3. Analysis classes and infrastructure classes serve as the basis for object-oriented elaboration.



- Traditional view is that a component (or module) resides in the software and serves one of three roles
    1. **Control components** coordinate invocation of all other problem domain components
    2. **Problem domain** components implement a function required by the customer
    3. **Infrastructure components** are responsible for functions needed to support the processing required in a domain application



Class-based Component Design

- Focuses on the elaboration of domain specific **analysis classes** and the **definition of infrastructure classes**
- Detailed description of **class attributes**, **operations**, and **interfaces** is required prior to beginning construction activities

Class-based Component Design Principles (SOLID)

- Modular - components by their very nature are **self-contained units of functionality** that are accessed and used in very specific ways.
- Cohesive - components are meant to address a **specific set of goals**, normally small in number, and everything within them supports the component achieving that goal or goals.
- Reusable - components aren't designed for just one project. Rather, they are designed to be used over and over, in any project that has a similar, or related need.
- Well Documented

Cohesion (lowest to highest)

Cohesion in software engineering is the degree to which the elements of a certain **module belong together**. It is a measure of functional strength of a module.

A good module, various parts having **high cohesion** is preferable due to its reliability, reusability, robustness and understandability. Low cohesion is associated with **undesirable traits** including **difficulty in maintaining**, **reusing** and **understanding.**

- **Logical cohesion** – when parts of a module are grouped because they are **logically categorized to do the same thing**, even if they are **different by nature**.
    - Print Functions: An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

- **Procedural cohesion** – is when parts of a module are grouped because they **always follow a certain sequence of execution** and commonly found at the top of hierarchy such as the main program

- A function which checks file permissions then opens the file

Excel file -extract -> web site

- **Communicational cohesion** – A module is said to have communicational cohesion, if all functions of the module refer to or update the **same data structure**.
  - the set of functions defined on an array or a stack.
  - Module determines customer details like use customer account no to find and return customer name and loan balance.
- **Sequential cohesion** is when parts of a module are grouped because the output from one part is the input to another part like an assembly line (e.g. a function which reads data from a file and processes the data).
  - Sort function -> search function -> display function
- **Functional cohesion** – module performs **one and only one function**
  - Compute cosine of angle
  - Read transaction record
  - Assign seat to airline passenger

Coupling (lowest to highest)

Coupling simply means to connect two or more things together. It means the pairing of two things. It actually measures the degree of independence between two things and how closely two things are connected or represent the strength of the relationship between them.

Module coupling means to couple two to more modules with each other and with the outside world. It generally represents how the modules are connected with another module and the outside world.

Coupling is related to cohesion. Cohesion means that the cohesive module performs only one task or one thing in the overall software procedure with a small amount of interaction with other modules. With the help of cohesion, data hiding can be done.

Low coupling correlates with high cohesion and high coupling correlates with low cohesion. Lower will be the coupling, higher will be the cohesion and better will be the program and these programs can be said as functionally independent of other modules.

- **Data Coupling:** Two modules are data coupled if they communicate through a parameter. An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc. This data item should be problem-related and not used for the control purpose.

```
class Receiver
 {
 public void message( MyType  X )
   {
   // code goes here
   X.doSomethingForMe( Object data );
   // more code
   }
 }
```

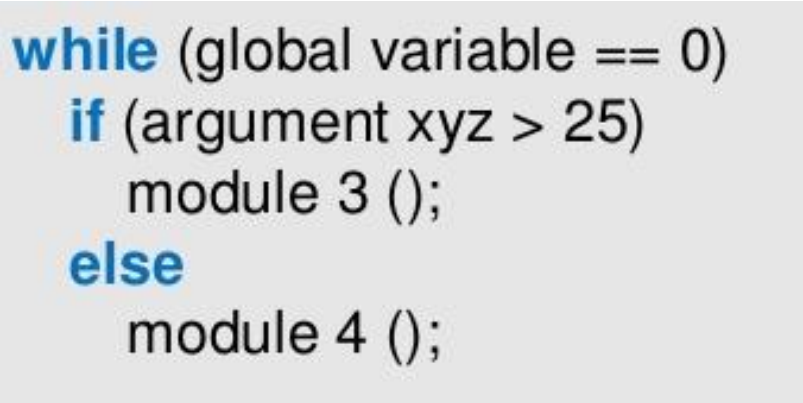
- **Stamp Coupling:** Two modules are stamp coupled if they communicate using a composite data item such as a record in PASCAL or a structure in C.

```
class A{
   // Code for class A.
};

class B{
    // Data member of class A type: Type-use coupling
    A var;

    // Argument of type A: Stamp coupling
    void calculate(A data){
        // Do something.
    }
};
```

- **Control Coupling:** Control coupling exists between two modules if data from one module is used to direct the order of instructions executed in another. An example of control coupling is a flag set in one module and tested in another module.

```
bool foo(int x){
    if (x == 0)
        return false;
    else
        return true;
}

void bar(){
    // Calling foo() by passing a value which controls its flow:
    foo(1);
}
```

- **Common Coupling:** Two modules are common coupled if they share data through some global data items.



- **Content Coupling:** Content coupling exists between two modules if one component references contents of another.
    - Component directly modifies another's data.
    - Component refers to local data of another component in terms of numerical displacement
    - Component modifies another's code, e.g., jumps into the middle of a routine

```
// tight coupling :

public int sumValues(Calculator c){
    int result = c.getFirstNumber() + c.getSecondNumber();
    c.setResult(result);
    return c.getResult();
}

// loose coupling :

public int sumValues(Calculator c){
    c.sumAndUpdateResult();
    return c.getResult();
}
```

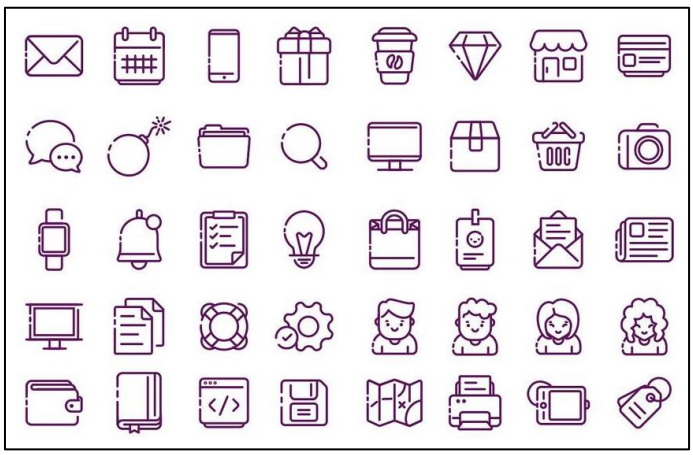
II. User interface design software engineering

User interface design or UI design generally refers to the visual layout of the elements that a user might interact with in a website, or technological product. This could be the control buttons of a radio, or the visual layout of a webpage. User interface designs must not only be attractive to potential users, but must also be functional and created with users in mind.

- System users often judge a system by its interface rather than its functionality.
- A poorly designed interface can cause a user to make catastrophic errors.
- Poor user interface design is the reason why so many software systems are never used.
- Most users of business systems interact with these systems through graphical interfaces although, in some cases, legacy text-based interfaces are still used.

Windows

Multiple windows allow different information to be displayed simultaneously on the user's screen

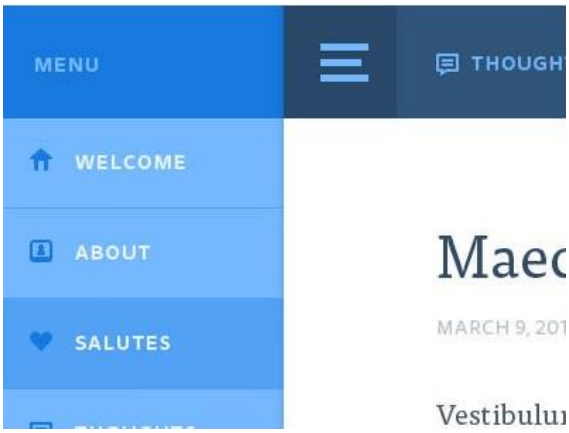


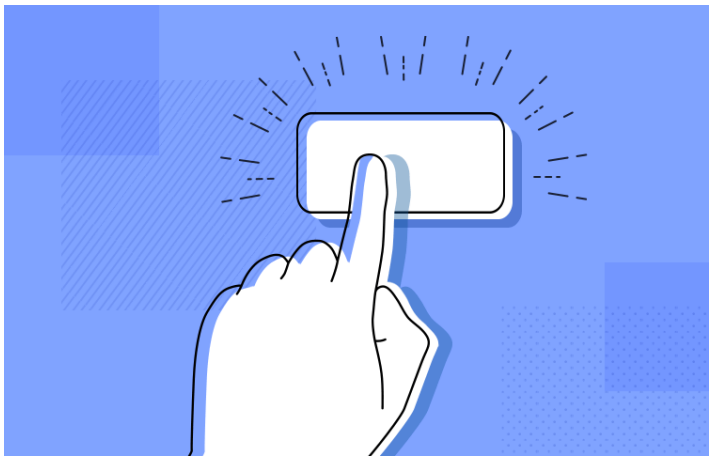
Icons

Icons different types of information. On some systems, it represents files; on others, it represents processes.

Menus

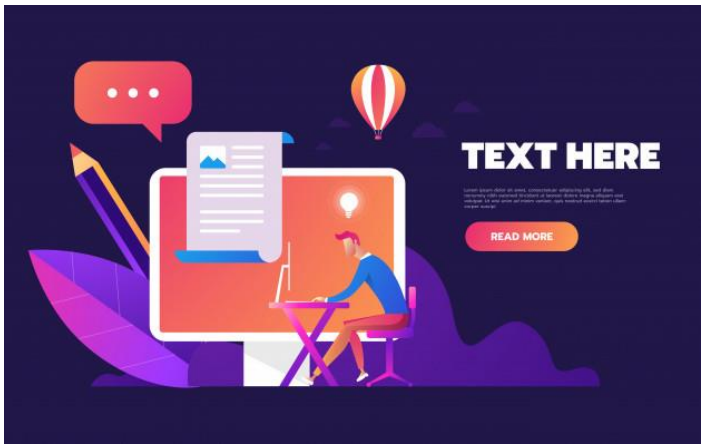
Commands are selected from a menu rather than typed in a command language





Pointing

A pointing device such as a mouse is used for selecting choices from a menu or indicating items of interest in a window.

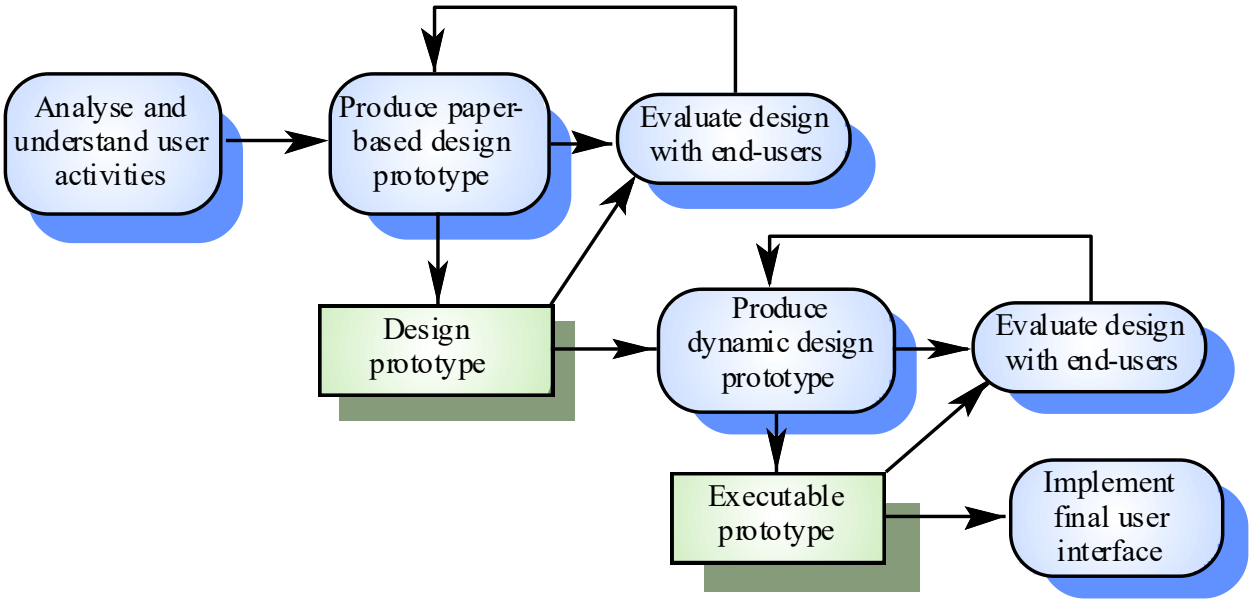


Graphics

Graphical elements can be mixed with text on the same display.

GUI advantages

- They are easy to learn and use.
    - Users without experience can learn to use the system quickly.
- The user may switch quickly from one task to another and can interact with several different applications.
    - Information remains visible in its own window when attention is switched.
- Fast, full-screen interaction is possible with immediate access to anywhere on the screen



UI design principles

- UI design must take account of the needs, experience and capabilities of the system users
- Designers should be aware of people's physical and mental limitations (e.g. limited short-term memory) and should recognize that people make mistakes
- UI design principles underlie interface designs although not all principles are applicable to all designs.

Design principles

- User familiarity
    - The interface should be based on user-oriented terms and concepts rather than computer concepts. For example, an office system should use concepts such as letters, documents, folders etc. rather than directories, file identifiers, etc.
- Consistency
    - The system should display an appropriate level of consistency. Commands and menus should have the same format, command punctuation should be similar, etc.
- Minimal surprise
    - If a command operates in a known way, the user should be able to predict the operation of comparable commands

- Recoverability
    - The system should provide some resilience to user errors and allow the user to recover from errors. This might include an undo facility, confirmation of destructive actions, 'soft' deletes, etc.
- User guidance
    - Some user guidance such as help systems, on-line manuals, etc. should be supplied
- User diversity
    - Interaction facilities for different types of user should be supported. For example, some users have seeing difficulties and so larger text should be available.

Interaction styles

Interaction style	Main advantages	Main disadvantages	Application examples
Direct manipulation	Fast and intuitive interaction Easy to learn	May be hard to implement Only suitable where there is a visual metaphor for tasks and objects	Video games CAD systems
Menu selection	Avoids user error Little typing required	Slow for experienced users Can become complex if many menu options	Most general-purpose systems
Form fill-in	Simple data entry Easy to learn	Takes up a lot of screen space	Stock control, Personal loan processing
Command language	Powerful and flexible	Hard to learn Poor error management	Operating systems, Library information retrieval systems
Natural language	Accessible to casual users Easily extended	Requires more typing Natural language understanding systems are unreliable	Timetable systems WWW information retrieval systems

CHAPTER **5**

SOFTWARE TESTING AND DEPLOYMENT

OBJECTIVES

At the end of the chapter, the student will be able to:

1. *distinguish the importance of V-model;*
2. *apply V-model in different software applications;*
3. *develop software application based on methods of software development;*
4. *understand the importance of reliability engineering;*
5. *use reliability engineering process in developing software application; and*
6. *perform network security*

I. SOFTWARE VERIFICATION AND VALIDATION

Software verification is a discipline of software engineering whose goal is to assure that software fully satisfies all the expected requirements. (Wikipedia)
Verification should check the program meets its specification as written in the requirements document for example.

Software verification tests (otherwise known as qualification tests) could include:

- testing of software to ensure that appropriate standards are met and that the software performs its intended functions, including audits of code (see below)
- ensuring system documentation is adequate and complete
- verifying that systems are capable of performing under expected normal conditions and possible abnormal conditions
- ensuring that security measures are in place and that they conform to appropriate standards
- ensuring that appropriate quality assurance measures are in place

Audits of software code may need to be conducted, particularly where the software is being used for a crucial system. Software audits are generally most effective when carried out by experts who are independent of the authors of the code. Measures included in a software audit could include:

- verifying that the code is logically correct
- ensuring the code is of modular design (that is, that the code is made up of discreet modules that can be separately tested and evaluated)
- verifying there is no 'hidden' code intended to perform unauthorized functions
- checking that the code is straightforward and relatively easy to understand
- ensuring the code is designed for easy testing - that is, that it includes features to allow testing of flow of data within and between modules
- verifying that the code is robust, so that it includes error trapping and error correction features that will allow immediate detection of errors and prevent loss of data through error

- ensuring the code incorporates security features that will prevent unauthorized access and/or detect and control any attempts at unauthorized access
- ensuring that the system is useable without the need for complex or obscure procedures
- ensuring that the software can be easily installed in the live environment
- verifying that the software can be easily maintained, and that errors or defects can be easily identified, corrected and validated after installation
- checking whether the software can be easily modified to add new features

Software Validation is a process of evaluating software product, so as to ensure that the software meets the pre-defined and specified business requirements as well as the end users/customers' demands and expectations.

Steps on conducting software validation

Step 1: Create the Validation Plan

The first step in the validation process is to create a validation plan (VP) that identifies who, what, and where. The plan usually consists of a description of the system, environment specifications, assumptions, limitations, testing criteria, acceptance criteria, identification of the validation team including the responsibilities of each individual, required procedures, and required documentation.

Step 2: Define System Requirements

The next step is defining the system requirements (SRS) defining what you expect the system to do. These can be broken down into two categories – infrastructure requirements and functional requirements. Infrastructure requirements include the identification of personnel resources needed and the facility and equipment needed for the task. Functional requirements include things such as the performance requirements, security requirements, system and user interfaces, and the operating environment needed.

Step 3: Create the Validation Protocol & Test Specifications

The testing phase begins with the development of a test plan (VP-Validation Protocol) and test cases (Test Specifications). The test plan describes the objectives, scope, approach,

risks, resources, and schedule of the software test. It documents the strategy that will be used to verify and ensure that a product or system meets its requirements.

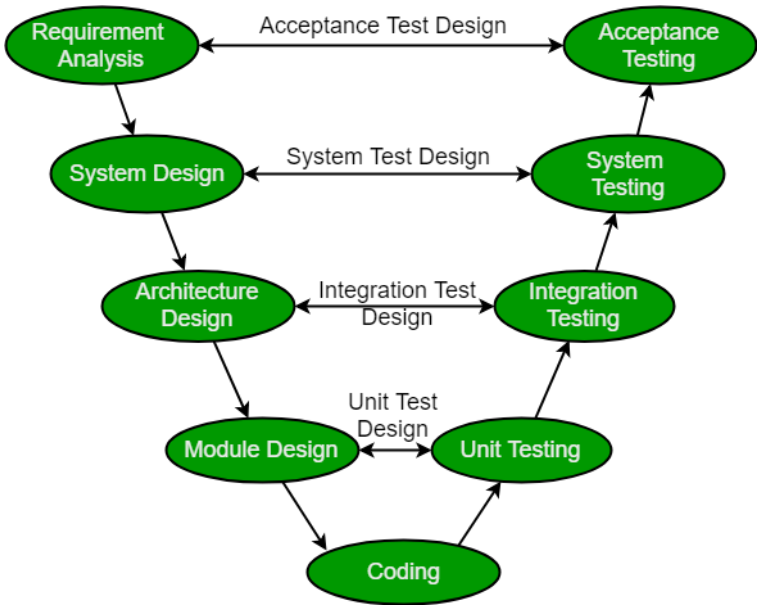
Step 4: Testing

The actual testing is then ready to be initiated. Black box testing (testing that ignores the internal code of the system or component and focuses on the inputs and outputs of the software) is used for validation of commercial off the shelf systems since you don't own the code. Tests are executed based on the test plan and test cases. Any errors, defects, deviations, and failures are identified and recorded, and are dispositioned in a final report.

Step 5: Develop/Revise Procedures & Final Report

Once testing is completed, procedures for system use and administration must be developed/revised. Then, prior to the system being released for use, a final validation report is produced, reviewed, and approved.

SDLC V-Model



The V-model is a type of SDLC model where process executes in a sequential manner in V-shape. It is also known as Verification and Validation model. It is based on the association of a testing phase for each corresponding development stage.

Design Phase:

Requirement Analysis: This phase contains detailed communication with the customer to understand their requirements and expectations. This stage is known as Requirement Gathering.

System Design: This phase contains the system design and the complete hardware and communication setup for developing product.

Architectural Design: System design is broken down further into modules taking up different functionalities. The data transfer and communication between the internal modules and with the outside world (other systems) is clearly understood.

Module Design: In this phase the system breaks down into small modules. The detailed design of modules is specified, also known as Low-Level Design (LLD).

Testing Phases:

Unit Testing: Unit Test Plans are developed during module design phase. These Unit Test Plans are executed to eliminate bugs at code or unit level.

Integration testing: After completion of unit testing Integration testing is performed. In integration testing, the modules are integrated and the system is tested. Integration testing is performed on the Architecture design phase. This test verifies the communication of modules among themselves.

System Testing: System testing test the complete application with its functionality, inter dependency, and communication. It tests the functional and non-functional requirements of the developed application.

User Acceptance Testing (UAT): UAT is performed in a user environment that resembles the production environment. UAT verifies that the delivered system meets user's requirement and system is ready for use in real world.

II. DEPENDABILITY

For many computer-based systems, **the most important system property is the dependability** of the system. The dependability of a system reflects **the user's degree of trust** in that system. It reflects the extent of the user's confidence that it will operate as users expect and that it will not 'fail' in normal use. Dependability covers the related systems attributes of reliability, availability and security. These are all inter-dependent.

System failures may have **widespread effects** with large numbers of people affected by the failure. Systems that are not dependable and are unreliable, unsafe or insecure may be **rejected by their users**. The **costs of system failure** may be very high if the failure leads to **economic losses** or **physical damage**. Undependable systems may cause information loss with a **high consequent recovery cost**.

Causes of failure:

1) **Hardware failure**

   Hardware fails because of design and manufacturing errors or because components have reached the end of their natural life.

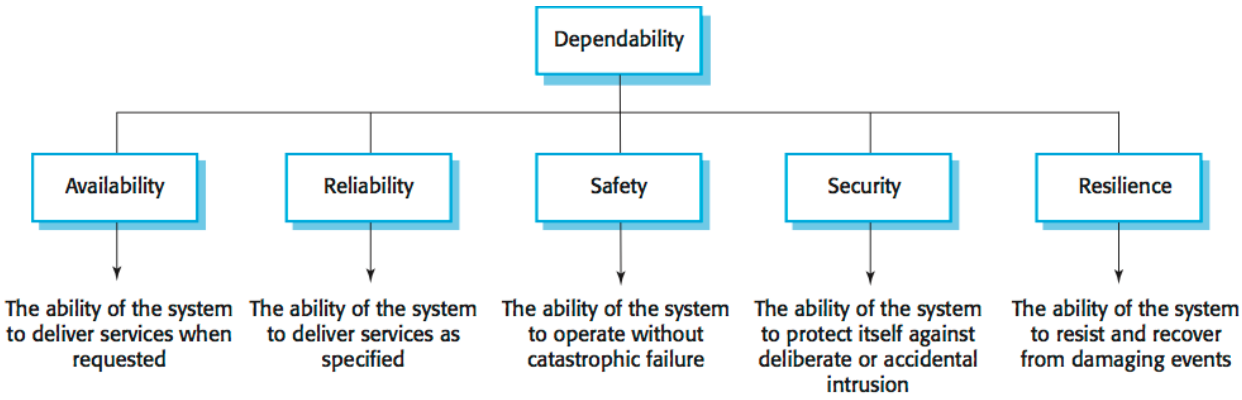
2) **Software failure**

   Software fails due to errors in its specification, design or implementation.

3) **Operational failure**

   Human operators make mistakes. Now perhaps the largest single cause of system failures in socio-technical systems.

Properties of Dependability



Principal properties:

- **Availability**: The probability that the system will be up and running and able to deliver useful services to users.
- **Reliability**: The probability that the system will correctly deliver services as expected by users.

- **Safety**: A judgment of how likely it is that the system will cause damage to people or its environment.
- **Security**: A judgment of how likely it is that the system can resist accidental or deliberate intrusions.
- **Resilience**: A judgment of how well a system can maintain the continuity of its critical services in the presence of disruptive events such as equipment failure and cyberattacks.

Other properties of software dependability:

- **Repairability** reflects the extent to which the system can be repaired in the event of a failure;
- **Maintainability** reflects the extent to which the system can be adapted to new requirements;
- **Survivability** reflects the extent to which the system can deliver services whilst under hostile attack;
- **Error tolerance** reflects the extent to which user input errors can be avoided and tolerated.

Many dependability attributes depend on one another. Safe system operation depends on the system being available and operating reliably. A system may be unreliable because its data has been corrupted by an external attack. Denial of service attacks on a system are intended to make it unavailable. If a system is infected with a virus, you cannot be confident in its reliability or safety.

How to achieve dependability?

- Avoid the introduction of accidental errors when developing the system.
- Design V & V processes that are effective in discovering residual errors in the system.
- Design systems to be fault tolerant so that they can continue in operation when faults occur.
- Design protection mechanisms that guard against external attacks.
- Configure the system correctly for its operating environment.
- Include system capabilities to recognize and resist cyberattacks.
- Include recovery mechanisms to help restore normal system service after a failure.

Dependability **costs tend to increase exponentially** as increasing levels of dependability are required because of two reasons. The use of more **expensive development techniques** and hardware that are required to achieve the higher levels of dependability. The **increased testing and system validation** that is required to convince the system client and regulators that the required levels of dependability have been achieved.

III. Formal methods of software development

Formal methods are intended to systematize and introduce rigor into all the phases of software development. This helps us to avoid overlooking critical issues, provides a standard means to record various assumptions and decisions, and forms a basis for consistency among many related activities. By providing precise and unambiguous description mechanisms, formal methods facilitate the understanding required to coalesce the various phases of software development into a successful endeavor.

The programming language used for software development furnishes precise syntax and semantics for the implementation phase, and this has been true since people began writing programs. But precision in all but this one phase of software development must derive from other sources. The term "formal methods" pertains to a broad collection of formalisms and abstractions intended to support a comparable level of precision for other phases of software development. While this includes issues currently under active development, several methodologies have reached a level of maturity that can be of benefit to practitioners.

Specific formalisms that will occupy our attention include:

- algebraic specification (including OBJ) -- used for specification and verification,
- predicate logic (including Z) -- used for specification and verification,
- state charts -- used for specification of "reactive" systems
- UML -- used primarily for design, and also for requirements analysis.

IV. Reliability engineering

Reliability engineering is engineering that emphasizes dependability in the life-cycle management of a product. Reliability is defined as the ability of a product or system to perform its required functions without failure for a specified time period and when used under specified conditions. Engineering and analysis techniques are used to improve the reliability or dependability of a product or system.

Reliability engineering falls within the maintenance phase of the software development life cycle (SDLC). The overall aim of the SDLC is to make software and products more reliable.

Reliability Engineering Objectives

The main objectives of reliability engineering are:

- To apply engineering knowledge and specialist techniques to prevent or to reduce the likelihood or frequency of failures
- To identify and correct the causes of failures that occur despite the efforts to prevent them
- To determine ways of coping with failures that occur, if their causes have not been fixed
- To apply methods for estimating the likely reliability of new software and for analyzing reliability data

Specification

Reliability can be difficult to specify, since it is defined in qualitative terms. Reliability is a prediction of the performance of a system or product in the future. It is usually defined as the probability that a product will operate without failure for a stated number of transactions over a stated period of time. Reliability metrics are stated as probability statements that are measurable by testing.

Examples of reliability requirements specifications are as follows:

- A patient monitoring system can fail less than 1 hour per year.
- The software shall have no more than x number of bugs per 1000 lines of code.

Implementation

The justification of reliability engineering hinges on the question: 'What is the cost of achieving reliability goals and the risk of not doing so?' Testing is the main implementation mechanism which can be used to determine the reliability of a system or product. Thorough testing, combined with the data it generates, product reliability predictions can be made. The measure of reliability is determined based on the trade-off between increased in levels of reliability and increased levels of testing. It important to note that failing tests prove the absence of reliability.

REFERENCES

(n.d.). Retrieved from ScienceDaily: https://www.sciencedaily.com/terms/computer_software.htm

Quilli, M. J. (2001, November 25). Retrieved from Object Oriented Analysis and Design: https://www.umsl.edu/~sauterv/analysis/488_f01_papers/quillin.htm

Software Engineering – Concepts and Implementations. (n.d.). Centre for Information Technology and Engineering.

Tutorialspoint. (n.d.). Retrieved from SDLC Overview: https://www.tutorialspoint.com/sdlc/sdlc_overview.htm

Bass, Len, Paul Clements, and Rick Kazman: Software Architecture in Practice, Second Edition. Boston: Addison Wesley, 2003. ISBN 0321154959.

Kruchten, Philippe. "Architectural Blueprints: The 4+1 View Model of Software Architecture." IEEE Software. 12 (6): 42-50.

Niquette, Paul. Softword: Provenance for the Word "Software." ISBN 1-58922-233-4.

(n.d.). Retrieved from Requirements Modeling in Software Engineering: Classes, Functions & Behaviors: https://study.com/academy/lesson/requirements-modeling-in-software-engineering-classes-functions-behaviors.html

Class-Based Data Modeling: Definition & Application. (n.d.). Retrieved from study.com: https://study.com/academy/lesson/class-based-data-modeling-definition-application.html

Glinz, M. (n.d.). *A Glossary of Requirements.*

OpenLearn. (n.d.). Retrieved from An introduction to software development: https://www.open.edu/openlearn/science-maths-technology/introduction-software-development/content-section-6

T., N. (n.d.). *Difference Between Entity and Attribute in Database*. Retrieved from binaryterms.com: https://binaryterms.com/difference-between-entity-and-attribute-in-database.html

Tutorialspoint. (n.d.). Retrieved from Software Requirements: https://www.tutorialspoint.com/software_engineering/software_requirements.htm

Citeseerx. (n.d.). Retrieved from Introduction to Software Reuse: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.7716&rep=rep1&type=pdf

study.com. (n.d.). Retrieved from Design Process in Software Engineering: Steps, Attributes & Changes: https://study.com/academy/lesson/design-process-in-software-engineering-steps-attributes-changes.html

study.com. (n.d.). Retrieved from Design Concepts in Software Engineering: Types & Examples: https://study.com/academy/lesson/design-concepts-in-software-engineering-types-examples.html

Thakur, D. (n.d.). *Computer Notes*. Retrieved from Architectural Design in Software Engineering: https://ecomputernotes.com/software-engineering/architecturaldesign

Geekforgeeks. (2020, June 16). Retrieved from Module Coupling and Its Types: https://www.geeksforgeeks.org/module-coupling-and-its-types/

Software Engineering. (2017, July 23). Retrieved from What is a Component? | An Object-Oriented View | The Traditional View | A Process-Related View: http://softwareengineeringmca.blogspot.com/2017/07/what-is-component-object-oriented-view-traditional-view-process-related-view.html

User Interface Design. (n.d.). Retrieved from Designing effective Designing effective: https://www.cs.umd.edu/~atif/Teaching/Spring2003/Slides/11.pdf