20. Ramez Elmasri and Shamkant B. Navathe, Fundamentals of Database Systems (Addison-Wesley, Reading, MA, 6th ed., 2006).

21. Thomas Connolly and Carolyn Begg, Database Systems: A Practical Approach to Design, Implementation, and Management (Addison-Wesley, Reading, MA, 5th ed., 2005).

22. Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom, Database Systems: The Complete Book (Prentice Hall, Upper Saddle River, NJ, 2nd ed., 2008).

SOFTWARE ENGINEERING AND OBJECT-ORIENTED ANALYSIS AND DESIGN (ACtE08)

8.1 SOFTWARE PROCESS AND REQUIREMENTS

Software:

Computer software refers to the piece of code, programs and routines that tell a computer what to do and how to do it. Software is a set of instructions that tell the computer hardware how to perform specific tasks. Examples of software include operating systems, productivity tools, games, and utilities.

A S/W is:

An instruction: When executed, provide desired features, functions and performances.

A data structure: Enables programs to adequately manipulate the information.

A document: Is a written description about how to use a software.

Software Engineer:

A software engineer is a professional who designs, develops, tests, and maintains software systems.

Software Engineering:

Software engineering is the process of designing, building, testing, and maintaining software systems in an organized, systematic, and efficient manner.

Types of Software:

- **System software:**

  It refers to the low-level software that manages and controls the operations of a computer, including the hardware and system resources. Examples of system software include:

  - **Operating Systems:** Windows, macOS, Linux, etc.
  - **Device Drivers:** Software that allows the operating system to communicate with and control hardware devices, such as printers, graphics cards, and network adapters.

- **Utilities:** Small, specialized programs that perform specific system-level tasks, such as disk defragmentation, compression, and backup.

- **Utility software:**
  It is a type of system software that provides essential tools for maintaining and optimizing a computer's performance. Examples of utility software include:
  - **Disk Cleanup:** Software that deletes temporary files and other unnecessary data from a computer's hard drive.
  - **Anti-virus/Anti-malware:** Software that protects a computer from malicious software and virus attacks.
  - **Backup and Restore:** Software that creates copies of important data and files, allowing them to be restored if lost or damaged.

- **Application software:**
  It refers to software that is designed to perform specific tasks and meet specific user needs. Examples of application software include:
  - **Office productivity software:** Microsoft Office, LibreOffice, etc.
  - **Graphic design software:** Adobe Photoshop, GIMP, etc.
  - **Web browsers:** Google Chrome, Mozilla Firefox, etc.
  - **Video editing software:** Adobe Premiere, Final Cut Pro, etc.
  - **Music and audio software:** Apple GarageBand, FL Studio, etc.

Software Characteristics:

- Comprised with two different roles:
  - Software is a product
  - Software is also a vehicle for delivering a product
- Logical not physical
- Developed or engineered, not manufactured.
- Doesn't wear out but deteriorate
- Usually, custom built

Software Quality Attributes:

Software quality attributes refer to the characteristics or features of software that determine its overall quality and usefulness to its intended users. Some common software quality attributes include:

- **Functionality:** The software should meet the requirements and provide the intended functions and features.
- **Usability:** The software should be easy to use and understand for its intended users.
- **Performance:** The software should be fast and efficient, with acceptable response times and minimal errors.

- **Reliability:** The software should be dependable and able to perform correctly even under adverse conditions.
- **Security:** The software should protect data and systems from unauthorized access and protect against potential threats.
- **Maintainability:** The software should be easy to modify, update, and extend as needed.
- **Portability:** The software should be able to run on different platforms and environments.
- **Scalability:** The software should be able to handle increased loads and demand over time.

Software Process Model:

1. **Plan Driven:**
   - Waterfall Model
   - Iterative Development
   - Prototype Model
   - Incremental Development
   - Spiral Development
   - Rapid Application Development (RAD) Model
2. **Agile:**
   - Scrum Model
   - Extreme Programming (XP)
   - Dynamic System Development Method
   - Kanban
   - Feature Driven Development
   - Lean Software Development
   - Crystal Methodologies

Software process model/ Software Development Lifecycle (SDLC)

SDLC is a methodology in which processes are defined to create the high quality software.

Phases of the SDLC:

- **Analysis:**
  Task performed during Analysis phase:
  - User research (what system user wants?)
  - Competitors Analysis
  - Business goal Establish
  - Budget planning
- **Design:**
  Task performed during Design phase:
  - Define product functionality
  - UI/UX designer creates Interface
  - Specify Product quality

- **Development:**
  Task performed during Development phase:
  - Programmer uses software design description by choosing any programming language to code modules.
  - Frontend development
  - Backend development
- **Testing:**
  Task performed during Testing phase:
  - Quality Assurance, Quality Control and testing performed before mass production testing.
  - Quality Assurance (QA): Finding errors, system re-engineer.
  - Quality Control (QC): Writing technical reviews, software testing, code inspections.
- **Deployment:**
  Task performed during Deployment phase:
  - Running the product into the server system.
- **Maintenance:**
  Task performed during Maintenance phase:
  - Keeps Software stable and up to date.
  - New bugs and vulnerabilities are solved.

Waterfall Model:

The Waterfall model is a sequential software development approach in which each stage of the process must be completed before moving on to the next. It follows a linear, step-by-step process with phases such as requirements gathering, design, implementation, testing, and maintenance. The model emphasizes documentation and is best suited for projects with well-defined requirements and predictable outcomes. However, it has limited flexibility and does not accommodate changes easily.

Phases of Waterfall model:

- Requirement Analysis
- System Design
- Implementation/ Coding
- Testing
- Deployment
- Maintenance

Advantages:

- Easy to understand and develop the model
- Documentation is prepared in each phase

Disadvantages:

- Difficult for a requirement change if process is underway. One phase must be completed to move to the next phase
- Uni- directional

Incremental Developmental Model:

Incremental development is a software development approach in which a product is developed in small, incremental releases. Each release builds upon the previous one to create a fully functional product. It allows for changes and feedback to be incorporated throughout the development process, making it more flexible than the Waterfall model. Customers are involved throughout the development process, and the model is well suited for agile development where requirements may change frequently. Incremental development helps manage risk by allowing for the evaluation and testing of each increment before proceeding with further development.

Analysis → Design → Coding → Testing — n^{th} increment

Analysis → Design → Coding → Testing — 2^{nd} increment

Analysis → Design → Coding → Testing — 1^{st} increment

Phases of incremental model are:

- Requirement Analysis
- Modelling
- Implementation
- Deployment

Advantages:

- Software objectives and requirements are met 100 %
- Client can provide feedback after each iteration

Disadvantages:

- Delay in delivery of the product if proper planning is not done
- Product cost increases if user frequently changes its requirement

Iterative Model:

The iterative software development life cycle (SDLC) is a software development methodology that emphasizes the repetition of a series of steps, or iterations, in order to gradually refine and improve a product or system. In iterative model, where a minimum viable product (MVP) is delivered early in the process and then improved upon through a series of cycles of development, testing, and feedback. This approach allows for continuous improvement, flexible adaptation to changing requirements, and faster time-to-market. In the iterative SDLC, each iteration builds upon the previous one and includes the development of new functionality, the refinement of existing functionality, and the resolution of any issues or problems. The iterative SDLC can be contrasted with a traditional, linear SDLC approach, which follows a more sequential, predictable path.

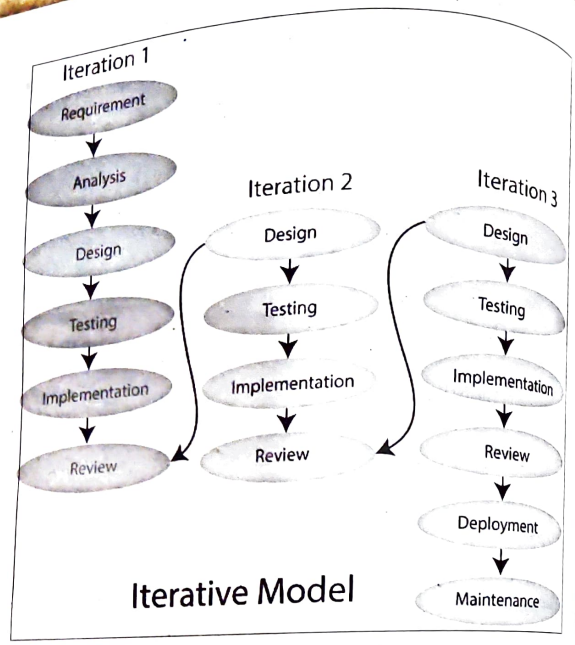


Figure: Iterative Model

Phases of iterative model:

- Requirement and Analysis phase
- Design
- Implementation
- Testing
- Deployment
- Review
- Maintenance

Advantages:

- Quick development of many features in SDLC
- Suitable for large projects

Disadvantages:

- Due to customer's frequent changing requirements, project completion date cannot be confirmed
- High cost due to risk management

Prototype Model:

The Software Prototype Model is a type of iterative software development process where a working model of the software is developed, tested, and then refined based on feedback from users or stakeholders. The prototype is essentially a preliminary version of the software that can be used to demonstrate the functionality and usability of the system to stakeholders.

The key benefits of the prototype model are that it allows developers to gather feedback from users and stakeholders early in the development process, making it easier to make changes and improvements as needed. It also helps stakeholders to better understand and visualize the end product, making it easier for them to provide meaningful feedback. The prototype model is often used when requirements are not well defined, or when the development team needs to experiment with different approaches before committing to a final design.

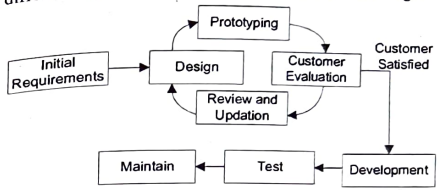


Figure: Prototype model

Advantages:

- User participates during the development phase so the project has less chance of fail
- Much earlier detection of errors

Disadvantages:

- User might have expectation of having final product's performance same as earlier prototype
- Confusion between final product and prototype

Big bang Model:

Big Bang SDLC (Software Development Life Cycle) is a non-iterative software development model where all phases of the development process are completed in one go, without any prototyping, iteration, or testing. In this model, the entire software development process is completed from start to finish without stopping to assess intermediate results. The Big Bang model is often used for simple projects with well-defined requirements and limited scope.

Figure: Big bang Model

Rapid Application Model (RAD):

The Rapid Application Development (RAD) model is an iterative and incremental software development process that emphasizes prototyping and rapid feedback. It involves developing software in small, manageable chunks, with a focus on user feedback and continuous improvement. The RAD model is designed to be fast and flexible, and it typically involves a team-based approach to software development. The goal of the RAD model is to deliver software quickly, while still ensuring that it meets the needs of the users and stakeholders.

Phases of RAD model:

- Business Modeling
- Process Modeling
- Testing and Turnover
- Data Modelling
- Application Generation

Advantages:

- Suitable when you have to reduce overall project risk.
- Requires less people but increases productivity.

Disadvantages:

- Highly skilled developers required.
- Only suitable for larger projects.

V Model:

The V-Model SDLC (Software Development Life Cycle) is a linear sequential model that emphasizes verification and validation throughout the software development process. It is represented as a "V" shape, with the two legs of the V representing the development and testing phases of the software. The V-Model is often used in government and military projects, where strict compliance with standards and regulations is required.

In the V-Model, each phase of the development process has a corresponding test phase, ensuring that the software is thoroughly tested and validated at every stage of development. This helps to catch and correct errors early in the development process, reducing the risk of costly rework later on. The V-Model is a step-by-step approach to software development that provides a clear, structured process for developing and testing software, and is well-suited for projects with well-defined requirements and a high degree of predictability.

a) Verification Phase
   - Business Requirement Analysis
   - Architectural Design
   - System Design
   - Module Design

b) Validation phase
   - Unit Testing
   - System Testing
   - Integration Testing
   - Acceptance Testing

Advantages:

- Preferred for smaller projects where requirements are too clear
- Easy to track progress

Disadvantages:

- Not recommended for Object oriented and complex projects
- No iteration of phases

Agile Model:

Agile SDLC (Software Development Life Cycle) is a flexible and iterative approach to software development that emphasizes rapid delivery of working software, collaboration between development teams and stakeholders, and the ability to respond to change. Agile development is based on the Agile Manifesto, which values individuals and interactions, working software, customer collaboration, and response to change over processes and tools.

In an Agile SDLC, the development process is divided into short cycles, known as sprints, each of which results in the delivery of a functional increment of the software. This allows for frequent feedback and adjustment based on changing requirements, and helps to ensure that the final product meets the needs of the stakeholders. Agile development is highly flexible and can accommodate changing requirements and priorities, making it well-suited for complex, rapidly changing projects. Additionally, Agile methodologies such as Scrum, Kanban, and XP have their own specific processes and practices, but all follow the core principles of Agile development.

Agile model is the combination of iterative and incremental process models. The steps involve in agile SDLC models are:

- Requirement gathering
- Design
- Unit testing
- Requirement Analysis
- Coding
- Acceptance testing

Advantages:

- Division of project into small iterations
- S/W development risk minimization

Disadvantages:

- Requires highly qualified developers
- New requirements might conflict/ not be appropriate with existing architecture

Computer Aided Software Engineering

Computer Aided Software Engineering (CASE) refers to the use of software tools to support the software development process. These tools assist developers in various tasks such as design, coding, testing, debugging, maintenance, and documentation. The main goal of CASE is to increase productivity, reduce errors and improve the quality of the software being developed.

Here are some examples of CASE tools:

- **Integrated Development Environments (IDEs):** These tools provide a comprehensive environment for software development, including a text editor, debugging tools, and a project manager. Examples include Visual Studio, Eclipse, and Xcode.

- **Code generators:** These tools automatically generate code based on specifications or templates, reducing the need for manual coding. Examples include Ruby on Rails scaffolding and Entity Framework Code First.
- **Modeling and design tools:** These tools support the creation of software models and designs, helping developers to visualize and understand the structure and behavior of the software being developed. Examples include Unified Modeling Language (UML) tools such as Enterprise Architect and Rational Rose.
- **Version control systems:** These tools manage and track changes to the source code of a software project, making it easier for multiple developers to collaborate on the same codebase. Examples include Git, Mercurial, and Subversion.
- **Test case management tools:** These tools assist in the management and organization of test cases and their execution. Examples include JIRA, Microsoft Test Manager, and TestRail.

Functional and Non-functional requirements:

Functional requirements specify the behaviors and tasks that a software system must perform, such as "The system must allow users to log in," or "The system must calculate the total cost of a shopping cart." These requirements describe what the system does and what its outputs are.

On the other hand, non-functional requirements specify the constraints and quality attributes of the system, such as "The system must be secure," or "The response time must be less than 2 seconds." These requirements describe how the system should behave and what its characteristics are.

Examples of functional requirements (fr):

- The system must allow users to search for and view product information.
- The system must generate reports based on user-specified criteria.
- The system must support multiple payment methods, including credit card and PayPal.

Examples of non-functional requirements (nfr):

- The system must be available 99.5% of the time.
- The system must be able to handle 1000 concurrent users.
- The system must comply with data privacy regulations, such as GDPR.
- The system must be user-friendly and accessible to people with disabilities.

NFRs are also known as software quality attributes. A model for classifying a software quality attribute is FURPS+

- F= Functionality: includes feature sets, capabilities and security.
- U= Usability: includes human factors, aesthetics, documentation.
- R= Reliability: frequency of failures, recoverability, predictability, accuracy.
- P= Performance: Processing speed, response time, resource consumption, throughput.

S= Supportability: Testability, extensibility, adaptability, maintainability, compatibility, configurability.

What's with the + in FURPS+?

Implementation, Interface, Operations, Packaging, Legal.

Requirements Engineering:

Requirements Elicitation: Process of determining the actual requirements of customer and users via communication. Sometimes called requirements gathering.

Requirement Analysis: Identifying and verifying whether the user requested requirements are valid, legal, ambiguous, unclear, incomplete and therefore resolving these issues.

Requirement Specification: To ensure clarity, consistency, and completeness of the system requirements, formal or semiformal documentation is done.

Types of Requirements:

User Requirements: Services and operational constraints provided by system is written in natural languages with detailed diagrams. Written for customers.

System requirements: Defines what should be implemented so may be part of a contract between client and contractor.

Software specification: A detailed software description which can serve as a basis for a design or implementation. Written for developers

Requirement Engineering Process:

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

Requirement Elicitation Techniques:

- Interviews
- Questionaries
- Brain storming
- Prototyping
- Surveys
- Task Analysis
- Domain Analysis
- Observation

Interface Specification:

An interface specification in software engineering is a detailed description of how two or more software components interact with each other. An interface defines a boundary between different components, specifying what each component can expect from the other components and what it needs to provide to them.

The interface specification defines the inputs and outputs of a component, as well as any error conditions that may occur. It also defines the methods or functions that a component exposes, and how other components can access those methods. In this way, the interface specification acts as a contract between the components, ensuring that they can work together in a predictable and consistent manner.

Examples of interface specifications include application programming interfaces (APIs), which define the interface between a software application and its underlying library or system components, and web services, which define the interface between a web application and other systems over the internet.

Requirement documents:

- Not a design document
- Tells WHAT the system should do rather than HOW it should do it.
- Requirement definition and specification must be included

IEEE requirements standard:

- Introduction
- General description
- Specific requirements
- Appendices
- Index

Requirements document structure:

- Introduction
- Glossary
- User requirements definition
- System architecture
- System requirements specification
- System models
- System evolution
- Appendices
- Index

Requirements validation techniques:

- **Requirements reviews**
  - Systematic manual analysis of the requirements.
- **Prototyping**
  - Using an executable model of the system to check requirements.
- **Test-case generation**
  - Developing tests for requirements to check testability.

Requirement's management:

At the time of requirement engineering and system development, the process of managing the customers frequently changing requirements.

8.2 SOFTWARE DESIGN

Software Design:

S/W design is the iterative process of translating user's requirement into a blueprint during the phase of software development. It is the process of defining the architecture, components, modules, interfaces, and data for a software system to satisfy specified requirements. It is a blue-print that guides the implementation of a software system.

Software design involves making trade-off decisions between competing factors such as functionality, performance, maintainability, scalability, testability, security, and other non-functional requirements.

The software design process can include activities such as requirements analysis, creating models and diagrams, selecting design patterns, defining data structures and algorithms, and specifying interfaces and APIs.

S/W design process goals:

- Analysis model contains all the explicit requirements, User's requirements also known as implicit requirements must be in the design.
- The developer team who codes the software and tests the software, they should easily read and understand the design.
- Behavioral, functional and data domain must be covered by design from an implementation view.

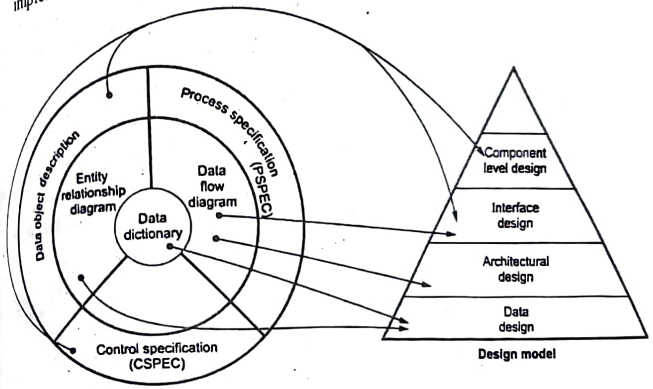


Figure: Analysis to design translation

Design Concepts:

There are several concepts that are commonly used in software design:

- **Abstraction:** Abstraction is the process of hiding complex implementation details and presenting a simplified view to the user. This helps in making the software design more modular and easier to understand.
- **Architecture:** Architecture refers to the overall structure of the software.
- **Information Hiding:** Hiding the unnecessary information i.e., procedure and data for those modules that doesn't require.

- **Encapsulation:** Encapsulation is the process of wrapping data and behavior together into a single unit called an object. This helps in protecting the data and behavior from being accessed or modified by unauthorized code.
- **Modularity:** Modularity refers to the practice of breaking a software system into smaller, independent, and interchangeable parts called modules. This makes the system easier to maintain, test, and extend.
- **Coupling and Cohesion:** Coupling refers to the degree to which one module or component depends on another, while cohesion refers to the degree to which elements within a module or component work together to achieve a single, well-defined purpose. In software design, it is important to strive for low coupling and high cohesion.
- **Separation of Concerns:** Separation of Concerns is the practice of dividing a software system into distinct parts, each of which addresses a separate concern or responsibility. This helps in making the software design more modular and maintainable.
- **Design Patterns:** Design patterns are pre-existing solutions to common design problems that have proven to be effective in many situations. They provide a common vocabulary and reusable solutions for software designers, making it easier to design and maintain complex software systems.
- **Reusability:** Reusability is the ability of software components to be used in multiple systems or applications. Designing for reusability helps in reducing development time and costs, as well as improving the quality and maintainability of the software.

Design Model:

- **Architecture Design:** The process of defining the high-level structure and organization of the software system, including the components, modules, and relationships between them.
- **Interface Design:** The process of defining the communication and data exchange between components, including APIs and protocols.
- **Data Design:** The process of defining the data structures and algorithms used by the software system, including data storage and retrieval.
- **Procedural Design:** Architecture's structural elements are transformed into s/w components procedural description.

S/W design heuristics:

Software design heuristics are guidelines and rules of thumb that help software designers make informed decisions during the design process. Some common software design heuristics include:

- **KISS (Keep It Simple, Stupid):** This heuristic encourages designers to simplify their designs as much as possible and to avoid overcomplicating the solution.
- **DRY (Don't Repeat Yourself):** This heuristic encourages designers to avoid redundancy in their designs and to reuse existing components and modules whenever possible.

- ***SOLID* (Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion):** This is a set of five principles that guide the design of object-oriented software and promote maintainability and scalability.
- ***YAGNI* (You Ain't Gonna Need It):** This heuristic encourages designers to avoid adding unnecessary features or components to their designs and to focus on delivering only what is essential.
- **High Cohesion, Low Coupling:** This heuristic encourages designers to aim for high cohesion within components and low coupling between components, to improve the maintainability and scalability of the software system.
- **Separation of Concerns:** This heuristic encourages designers to divide the software system into separate, well-defined components, each of which addresses a distinct concern or responsibility.
- **Information Hiding:** This heuristic encourages designers to encapsulate the implementation details of components and to expose only their public interfaces, to reduce the impact of changes and to improve the maintainability of the software system.

Design Principles:

- **Tunnel Vision:** Design process shouldn't suffer.
- **Traceable:** Design must be traceable to analysis model.
- **Wheel:** Design shouldn't reinvent wheel.
- **Intellectual distance:** Design must minimize it between S/W and the real problem.
- **Uniformity and integration:** Design must possess it.
- **Change:** Design must adapt to change.

S/W architectural design:

To build a computer-based system, it requires structure's data and program components which is represented by software architectural design.

- **Generic application:** Can we use any generic application architecture? Is it available?
- **Distribution:** How the system will be distributed?
- **Architectural styles:** Which architectural styles best suited?
- **Approach:** For structuring the system, which approach is preferred?
- **Decomposition:** How to decompose the system?
- **Control strategy:** Which control strategy to be used?
- **Evaluation:** How to evaluate the architectural design?
- **Documentation:** How should be architectural design be documented?

System Organisation:

System Organisation depicts the basic strategy on structuring a system.

Organizational styles are:

- Shared services and server styles
- Abstract machine or layered styles
- Shared data repository styles

Modular decomposition styles:

Decomposition of sub-systems into smaller fragments called modules.

Two modular decomposition models are:

- **Object model:** In which complete system is decomposed into interacting objects.
- **Data flow/ Pipeline model:** In which system is decomposed into functional modules which then transforms inputs to outputs.

Control flow styles:

Control flow controls the flow between many subsystems.

- **Centralized control:** One sub-system acts as a master, means which has control over other subsystems means master can start or stop other subsystems. Two models:
  - Call return model
  - Manager model
- **Event-based control:** Sub systems can generate events, each subsystem can interact/ respond to externally created events from other sub systems. Two principal event-driven models:
  - Broadcast model
  - Interrupt-driven model

Reference Architecture:

Reference architectures are not derived from the existing systems. They are usually derived from the research or study of application domain perspective. Reference architecture acts as a standard framework in which other architectures can rely on or other systems can be evaluated. E.g. OSI reference model for communication systems and whose actual implementation model is TCP/IP model.

Multi-processor Architecture:

Multi-processor architecture refers to computer systems that have multiple processors, or cores, working in parallel to perform tasks. The goal of multi-processor architecture is to increase the processing power and speed of a computer system by dividing complex tasks into smaller, more manageable subtasks that can be executed in parallel.

Client server Architecture:

A client-server architecture is a distributed computing model in which client devices connect to and request services or resources from servers over a network. The servers are responsible for providing the requested services or resources, while the clients are responsible for

requesting and using them. In a client-server architecture, the client device acts as the front-end interface, while the server acts as the back-end data repository and processing power. This architecture allows for the separation of responsibilities and provides a scalable and flexible solution for distributed computing. There are many different types of client-server architectures, including two-tier, three-tier, and n-tier architectures. In a two-tier architecture, the client and server are directly connected, while in a three-tier architecture, there is an additional middle-tier component that acts as an intermediary between the client and server. In an n-tier architecture, there are multiple middle-tier components that perform various functions and provide additional levels of abstraction and modularity.

Distributed object Architecture:

Distributed object architecture is a software design pattern that allows for the distribution of objects across multiple networked computers. In this architecture, objects are considered to be the basic units of software and are used to encapsulate data and behavior. The objects are then distributed across the network and can interact with each other through remote method invocations.

Distributed object architecture provides a way to build scalable and distributed applications by allowing objects to interact with each other as if they were running on the same machine. This architecture provides many benefits, including increased reliability, scalability, and performance. It also allows for the integration of multiple different programming languages and systems.

A common example of a distributed object architecture is the Common Object Request Broker Architecture (CORBA), which is a middleware system that enables communication between objects running on different computers. CORBA uses an object request broker (ORB) to manage the distribution of objects and the communication between them.

Inter-organizational distributed computing:

Inter-organizational distributed computing refers to the use of distributed computing techniques and technologies between organizations to collaborate and share computing resources. This type of distributed computing enables organizations to pool their computing resources and share data, applications, and services over a network.

In inter-organizational distributed computing, organizations can benefit from each other's computing resources and expertise, reducing the cost of ownership and increasing efficiency. For example, one organization may have excess computing resources that it can offer to another organization, while the second organization may have specialized expertise in a particular area that can be shared with the first organization.

Inter-organizational distributed computing also enables organizations to collaborate on complex projects that require a large amount of computing resources. By sharing resources, organizations can reduce the time and cost of projects and increase their competitiveness. Examples of inter-organizational distributed computing include cloud computing, grid computing, and collaborative scientific computing.

Real time software design:

Real-time software design refers to the development of software that is designed to respond to events and execute within strict time constraints. Here are some key points to consider when designing real-time software:

- **Determining Real-Time Requirements:** The first step in designing real-time software is to determine the real-time requirements of the application. This includes defining the time constraints and deadlines for the software, and identifying the events that the software must respond to.
- **Predictable Performance:** Real-time software must have predictable performance, meaning that its behavior must be consistent and reliable, even in the face of hardware or software failures.
- **Prioritizing Tasks:** Real-time software must prioritize tasks based on their deadlines and importance, so that critical tasks are completed within their required time constraints.
- **Concurrency and Multitasking:** Real-time software must be designed to handle multiple tasks simultaneously and must manage the concurrent execution of tasks to meet the required time constraints.
- **Error Handling:** Real-time software must have robust error handling mechanisms to ensure that it can continue to operate correctly even in the face of failures.
- **Testing and Validation:** Real-time software must be thoroughly tested and validated to ensure that it meets the real-time requirements and behaves as expected in all scenarios.
- **Scalability:** Real-time software must be designed to scale to meet the changing demands of the application, as the number of events and tasks increases over time.
- **Real-Time Operating System:** Real-time software may require a real-time operating system that provides low-latency, deterministic performance and real-time scheduling capabilities.

Component-based software engineering:

Component-based software engineering (CBSE) is a software development methodology that emphasizes the separation of an application into modular and reusable components. These components can be easily integrated into new or existing systems, making the development process more efficient, flexible, and scalable. CBSE can also lead to faster development time, as components can be reused across multiple projects, reducing the need to write the same code multiple times.

Some examples of CBSE in practice include:

- **Object-Oriented Programming (OOP):** In OOP, components are represented by objects that encapsulate data and behavior. For example, an object representing a car might have properties like make, model, and year, and methods like start, drive, and stop.
- **Service-Oriented Architecture (SOA):** In SOA, components are services that can be accessed and reused by different parts of a system. For example, a weather service that provides information about the current weather conditions could be used by multiple applications, such as a news app or a weather app.

- **Microservices:** Microservices are a type of SOA where each component is a small, independent service that performs a specific function. For example, a microservice-based e-commerce application might have separate microservices for handling user authentication, product catalog management, and order processing.
- **Component-based frameworks:** Component-based frameworks provide pre-built components that can be used to build an application. For example, Angular is a component-based framework for building web applications, where each component represents a piece of the user interface, such as a form or a button.

8.3 SOFTWARE TESTING, COST ESTIMATION, QUALITY MANAGEMENT, AND CONFIGURATION MANAGEMENT

Software testing

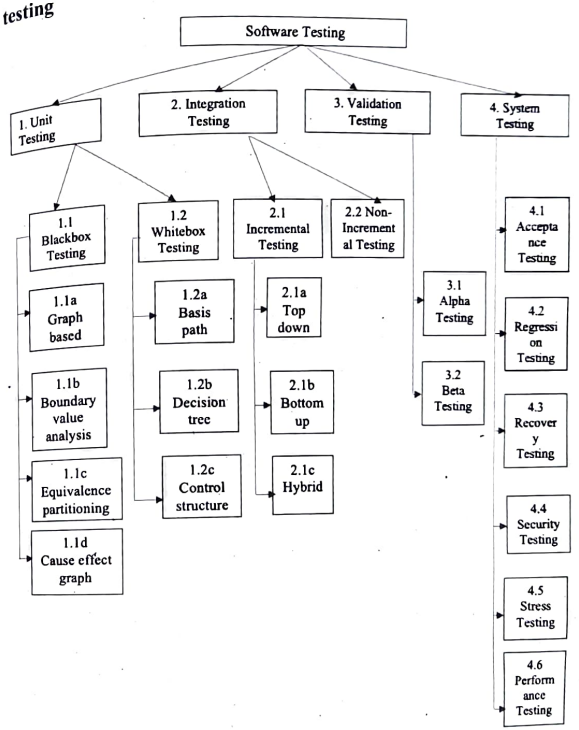


Figure: Classification of software testing

Software testing:

Software testing is the process of evaluating a software application or system to detect potential defects, errors, or bugs that could affect its functionality, reliability, or security. It involves a range of activities and techniques that aim to verify and validate the software's behavior against its intended requirements and user expectations, as well as to identify any unexpected or undesired outcomes. The goal of software testing is to ensure that the software works as expected, meets quality standards, and delivers value to its users.

- **Unit testing:** Unit testing is a type of testing that verifies the behavior and functionality of individual units or components of a software application, such as functions, methods, or classes. It is performed by developers and involves writing and executing automated test cases that target specific units of code in isolation from the rest of the system.
- **Integration testing:** Integration testing is a type of testing that focuses on verifying the interactions and compatibility between different components or modules of a software system. It is performed after unit testing and involves testing the integrated system to ensure that the individual components work together as intended and produce the expected output.
- **System testing:** System testing is a type of testing that verifies the behavior and functionality of a complete software system in a real-world environment. It is performed after integration testing and involves testing the system as a whole to ensure that it meets the specified requirements and user expectations.
- **Component testing:** Component testing is a type of testing that verifies the behavior and functionality of individual software components, such as libraries, frameworks, or services. It is performed during the development phase and involves testing each component separately to ensure that it works as intended and meets its requirements.
- **Acceptance testing:** Acceptance testing is a type of testing that verifies the behavior and functionality of a software system from the user's perspective. It is performed after system testing and involves testing the system in a production-like environment to ensure that it meets the user's requirements and business needs. Acceptance testing is usually performed by end-users or stakeholders and can take the form of manual or automated testing.

Unit Testing:

- **Black Box testing:** Blackbox testing is a software testing technique that examines the functionality of a software application without considering its internal workings or structure. Testers evaluate the system's input and output and compare the results with expected outcomes to determine if the software is working correctly. This type of testing focuses on the software's behavior, and the tester does not have any knowledge of the underlying code or internal logic.

- ***Graph-based testing***: It is a software testing technique that involves using graphs to model the behavior and interactions of a software system. Testers use these models to identify potential defects, coverage gaps, and other issues in the system.
- **Boundary value analysis**: It is a testing technique that focuses on testing values at the boundaries of the input domain. The objective of this technique is to identify any defects that may occur at the edge of the input range, which can help testers identify potential issues that may occur when the software is deployed in production.
- **Equivalence partitioning:** It is a software testing technique that involves dividing the input domain of a system into equivalent classes or partitions. Testers identify the input values that produce the same output, and test a representative value from each partition. This approach helps in reducing the number of test cases and improves the efficiency of the testing process.
- ***Cause-effect graphing:*** It is a testing technique that involves creating a graphical representation of the system's behavior based on its inputs and outputs. Testers identify the inputs that cause specific behaviors or outputs and use this information to create a cause-effect graph. This approach helps in identifying the most critical scenarios for testing and can help testers to identify and resolve defects more quickly.

White Box testing: Whitebox testing, also known as structural testing, is a software testing technique that evaluates the internal workings of an application, including its code, design, and internal logic. Testers use this technique to verify the correctness of the application's code, including the expected and unexpected interactions between different parts of the code. Whitebox testing is typically performed by developers who have knowledge of the system's internal architecture and implementation details. It can be performed at different levels of the software development process, including unit testing, integration testing, and system testing. Whitebox testing is used to ensure that the code is functioning as expected and to identify any defects or issues that may occur during the operation of the software. This technique is used in combination with other software testing techniques, such as Blackbox testing, to ensure the highest quality and reliability of the software application.

- **Basis path testing:** It is a white box testing technique used to test the flow of control within a software application. The technique involves creating a mathematical model of the application's control flow graph, which represents all possible sequences of execution. Testers then use this model to derive a set of test cases to ensure that all paths in the application are tested at least once.
- **Decision tree:** In white box testing, a decision tree is a graphical representation of all possible decision paths in a software application. The tree is constructed based on the application's decision-making logic, with each node representing a decision point and each branch representing a possible decision outcome. Testers use the decision tree to identify the most critical decision points in the application and to create test cases to ensure that each possible decision path is tested at least once.

- **Control structures:** It refer to the flow of control within a software application. In programming, control structures are used to determine the order in which statements are executed. Control structures include conditional statements (if/else), loops (for/while), and function calls. Testers use knowledge of the control structures in white box testing to ensure that the application is behaving as expected and to identify any defects or issues that may occur during execution.

Integration Testing:

- **Incremental testing:** In this approach, the software components are integrated incrementally, i.e., one by one, until all the components are integrated and tested together as a complete system. Each incremental step includes the integration and testing of a new module, and the previously integrated modules are retested to ensure that they are still functioning correctly.
  - **Top-Down Integration Testing:** In this approach, the testing starts from the top layer of the application, which is the user interface layer, and proceeds downwards to the lower layers. This type of testing is useful for applications with complex architectures.
  - **Bottom-Up Integration Testing:** In this approach, testing starts from the lower layers of the application and proceeds upwards to the top layer. This type of testing is suitable for applications with a large number of modules.
  - **Hybrid Integration Testing:** This approach combines both top-down and bottom-up integration testing approaches, where the testing is done in both directions to ensure that all the components of the software application are integrated and work together correctly.
- **Non-incremental testing:** In this approach, all the components of the software application are integrated and tested together as a complete system in one go, without any incremental steps. The advantages of non-incremental testing are that it provides a complete view of the system, and it ensures that all the components work together as intended. However, this approach is often riskier, as it is difficult to identify the root cause of defects, and fixing them can be time-consuming.

Validation Testing:

- **Alpha testing:** It is the preliminary testing carried out by developers to evaluate the software's overall functionality and identify any issues before releasing it to the general public. It's a kind of in-house testing where developers test the software before its launch.
- **Beta testing:** It is the second stage of testing, in which the software is tested by a group of users, known as beta testers, who are not part of the development team. The software is released to the beta testers, who then evaluate it in a real-world environment and provide feedback on the software's performance, usability, and overall user experience. The feedback received from beta testers is used to make the final adjustments and improvements to the software before it is released to the public.

Alpha testing vs Beta testing differences:

Some of the key differences between Alpha testing and Beta testing are as follows:

- **Participants:** Alpha testing is done by the software development team, while Beta testing is conducted by a group of users who are not a part of the development team.
- **Environment:** Alpha testing is done in-house, i.e., within the company's premises, while Beta testing is carried out in a real-world environment.
- **Objectives:** Alpha testing is focused on identifying and fixing bugs and errors in the software, while Beta testing is focused on evaluating the software's performance, usability, and user experience.
- **Timing:** Alpha testing is carried out before the software is released to the public, while Beta testing is done after the software's alpha testing phase is complete and before its official release.
- **Scope:** Alpha testing covers a limited number of users, whereas Beta testing is open to a larger group of users.
- **Feedback:** In Alpha testing, feedback is received from the software development team, while in Beta testing, feedback is received from end-users.

System testing:

- **Acceptance Testing:** This type of testing is carried out to verify whether the system meets the user requirements and whether it is ready for release. It is typically performed by end-users, stakeholders, and product owners.
- **Regression Testing:** This testing is done to ensure that the changes made to the system do not negatively affect the existing functionalities. It is usually done after every new release or update.
- **Recovery Testing:** This testing is done to check the system's ability to recover from hardware or software failures. It helps to ensure that the system can resume normal operations after an unexpected event or crash.
- **Security Testing:** This testing is done to identify and mitigate potential security risks and vulnerabilities in the system. It helps to ensure that the system is secure against cyber-attacks and unauthorized access.
- **Stress Testing:** This type of testing is carried out to assess the system's performance under high stress conditions. It helps to determine the system's breaking point and its ability to handle heavy loads and high traffic.
- **Performance Testing:** This testing is carried out to check the system's performance, speed, and responsiveness under normal or expected conditions. It helps to identify any bottlenecks, latency issues, and other performance-related problems in the system.

Test case and Test suit:

In software testing, a test case is a set of instructions, conditions, and inputs that are designed to verify the behavior and functionality of a specific feature, function, or requirement of a software application. A test case typically includes a description of the feature being tested, the expected output or behavior, and the specific input values or conditions that will be used to test the feature.

A test suite, on the other hand, is a collection or group of related test cases that are organized and executed together as a unit. A test suite can be used to test a specific module, component, or feature of a software application, or it can be used to test the entire system. The purpose of a test suite is to ensure that all the related test cases are executed efficiently and effectively, and to help ensure comprehensive testing of the software application.

Test case design:

Test case design is the process of creating and developing test cases that are designed to verify the behavior and functionality of a software application. Test case design involves identifying the specific features, functions, or requirements of the software application that need to be tested, and creating a set of test cases that cover all possible scenarios and use cases for those features and functions. There are various techniques and approaches to test case design, including boundary value analysis, equivalence partitioning, decision table testing, state transition testing, and exploratory testing. The selection of a test case design technique depends on the specific requirements and characteristics of the software application, as well as the goals and objectives of the testing process.

The key elements of test case design include:

- Test objectives and requirements
- Test steps and procedures
- Test execution and reporting
- Test inputs and expected outputs
- Test data and test environment

Effective test case design is critical for ensuring that a software application is thoroughly tested and meets the specified requirements and user expectations. Well-designed test cases can help detect defects and issues early in the development cycle, reduce costs and time-to-market, and improve the overall quality and reliability of the software product.

Approaches to test case design:

There are several approaches to test case design in software testing, some of which include:

- **Equivalence Partitioning:** This approach involves dividing the input values into classes or partitions that are expected to behave in a similar way. Test cases are then designed to represent each partition, ensuring that at least one test case is executed for each partition.
- **Boundary Value Analysis:** This approach involves testing the values that are at the boundaries of the input domain. Test cases are designed to ensure that the system behaves correctly when it receives values that are on the edge of the acceptable range.

- **Decision Table Testing:** This approach involves creating a table that specifies the inputs and expected outputs for a particular feature. Test cases are then designed to test each combination of inputs and outputs.
- **State Transition Testing:** This approach involves testing the system's behavior as it moves from one state to another. Test cases are designed to verify that the system behaves correctly as it transitions from one state to another.
- **Exploratory Testing:** This approach involves testing the system without a formal test plan or predefined test cases. The tester explores the system to identify defects and issues, and then creates test cases to address those issues.

Test automation:

Test automation is the use of specialized software tools and frameworks to automate the execution of test cases and the evaluation of test results in software testing. The goal of test automation is to increase the efficiency, speed, and accuracy of the testing process by reducing the manual effort required for repetitive and time-consuming testing tasks.

Test automation can be applied to various types of testing, including unit testing, integration testing, system testing, and acceptance testing. Test automation typically involves the use of programming languages and testing frameworks to develop and execute test scripts, which simulate the actions and behavior of a user or a system.

The benefits of test automation include:

- Improved testing efficiency and speed
- Increased test coverage and accuracy
- Reduced testing costs and time-to-market
- Improved reliability and quality of the software application
- Enhanced collaboration and communication among team members

However, test automation also has some limitations and challenges, such as the need for specialized skills and knowledge, the cost and effort required to develop and maintain test scripts, and the limitations of test automation tools in testing certain types of software applications.

Software testing metrics:

Software testing metrics are used to measure and evaluate the effectiveness, efficiency, and quality of the software testing process. They are used to provide objective and quantitative data on various aspects of testing, such as the number of defects found, the time and effort required for testing, the test coverage and quality, and the progress and status of the testing process.

Some common software testing metrics include:

- **Defect Density:** This metric measures the number of defects found per unit of code or requirements. It can help identify areas of the software application that are more error-prone and require more testing effort.

- **Test Coverage:** This metric measures the extent to which the software application has been tested, based on the requirements and features that have been tested. It can help identify gaps in test coverage and ensure that all critical features and functions have been tested.
- **Test Execution Time:** This metric measures the time required to execute all the test cases in the test suite. It can help identify areas where testing can be optimized and improved to reduce testing time and effort.
- **Test Pass Rate:** This metric measures the percentage of test cases that have passed successfully. It can help track the progress and effectiveness of the testing process and identify areas that need improvement.
- **Defect Rejection Rate:** This metric measures the percentage of defects that have been rejected or not accepted by the development team. It can help identify areas where defects are not being reported effectively or are not being resolved in a timely manner.

Types of test metrics:

- **Process metrics:** improves process efficiency of SDLC.
- **Product metrics:** improves software product quality.
- **Project metrics:** measures efficiency of projects team members and also measures the efficiency of tools used by the team members for testing purpose.

Algorithmic cost modelling:

Algorithmic cost modeling is a technique used in software engineering to estimate the cost, effort, and resources required to develop a software application. It involves the use of mathematical models, algorithms, and historical data to predict the time and cost required to develop a software application, based on various factors such as the size and complexity of the application, the development methodology, the programming languages and tools used, and the skills and experience of the development team.

There are several algorithmic cost modeling techniques that are commonly used in software engineering, including:

- **Function Points:** This technique measures the functionality of a software application in terms of the number and complexity of the user interactions and data processing operations, and uses a set of standardized conversion factors to estimate the effort and cost required to develop the application.
- **COCOMO:** The Constructive Cost Model (COCOMO) is a widely used algorithmic cost modeling technique that estimates the effort required to develop a software application, based on its size and complexity, and factors such as the development methodology, the programming languages and tools used, and the experience and productivity of the development team.
- **SLIM:** The Software Life-cycle Management (SLIM) model is a comprehensive cost estimation and project management tool that uses historical data and statistical analysis to estimate the cost, effort, and duration of a software development project, and to manage project risks and resources.

Project duration and staffing:

Project duration and staffing are important factors in software engineering that can have a significant impact on the success of a software development project. Here are some key considerations for project duration and staffing:

- **Project Duration:** The duration of a software development project is the total time required to complete all the phases of the project, from requirements analysis to delivery and maintenance. It is important to estimate the project duration accurately, taking into account factors such as the complexity of the application, the scope of the project, the development methodology, and the availability of resources.
- **Staffing:** Staffing refers to the number and type of resources required to complete the project within the given timeline. It is important to identify the necessary skills and expertise required for each phase of the project, and to ensure that the team has the necessary resources to complete the project on time and within budget.

Software Quality:

Software Quality refers to the degree to which a software product meets the specified requirements, customer expectations, and industry standards. It is an essential aspect of software development and involves testing and verifying the software's functionalities, performance, reliability, usability, security, and maintainability. Good software quality ensures that the software is free of defects, meets the customer's needs, and performs its intended functions effectively and efficiently.

Qualities of a good software:

Some of the representative qualities of software are:

- **Functionality:** The software should be designed to meet the user requirements and perform its intended functions effectively.
- **Reliability:** The software should be reliable, i.e., it should perform consistently and accurately over time, even under different operating conditions.
- **Usability:** The software should be easy to use and user-friendly, with a clear and intuitive interface.
- **Maintainability:** The software should be easy to maintain, update, and modify as required without affecting the overall functionality.
- **Efficiency:** The software should be efficient in terms of using the system resources effectively and providing fast response times.
- **Portability:** The software should be portable, i.e., it should be able to run on different operating systems and platforms without any issues.
- **Security:** The software should be secure, i.e., it should protect the system and user data against unauthorized access and malicious attacks.
- **Compatibility:** The software should be compatible with other software and hardware systems to ensure interoperability and smooth functioning.

SQA:

Software Quality Assurance is the process of ensuring that software products and processes comply with established standards and best practices. SQA involves a set of activities and processes aimed at improving the quality of software development, testing, and deployment. It involves the use of methodologies, processes, and tools to monitor and improve the software development process and ensure that the software product meets the expected quality standards. The key objective of SQA is to reduce software defects, improve the quality of the software product, and increase customer satisfaction.

SQA Elements:

The main elements of Software Quality Assurance (SQA) are as follows:

- **Software Development Process:** SQA ensures that the software development process adheres to established standards, best practices, and guidelines. It involves defining the processes, methods, and tools used to develop, test, and maintain the software.
- **Software Testing:** SQA involves testing the software to ensure that it meets the specified requirements, performs as intended, and is free of defects. Testing involves different types of testing, including functional testing, regression testing, security testing, and performance testing.
- **Documentation:** SQA requires documentation of the software development process, including requirements, design, and testing documents. Proper documentation ensures that the software can be maintained and updated by different team members.
- **Training and Education:** SQA involves training and educating team members about the software development process, methodologies, and tools used. This helps to ensure that team members have the required skills and knowledge to develop high-quality software.
- **Metrics and Measurements:** SQA involves collecting and analyzing metrics and measurements to monitor the software development process's performance. Metrics help to identify the areas for improvement and to track progress towards the established quality goals.
- **Continuous Improvement:** SQA involves continuously improving the software development process by identifying areas for improvement, implementing corrective actions, and monitoring the effectiveness of the improvements made.

SQA Tasks:

The main tasks of Software Quality Assurance (SQA) include:

- Developing and implementing software development processes, procedures, and standards to ensure the software is developed according to industry best practices.
- Defining testing strategies, methodologies, and techniques to ensure that the software meets the specified requirements and performs as intended.
- Establishing metrics, collecting data, and analyzing it to monitor the software development process's performance, identify areas for improvement and track progress towards quality goals.

- Documenting the software development process, including requirements, design, and testing documents, to ensure that the software can be maintained and updated by different team members.
- Conducting audits and reviews of the software development process to identify and correct deficiencies, non-compliance with established standards, and potential areas for improvement.
- Training and educating team members on the software development process, methodologies, and tools used to ensure that team members have the required skills and knowledge to develop high-quality software.
- Collaborating with other teams and stakeholders to ensure that the software development process aligns with the overall project objectives and business goals.
- Continuously improving the software development process by identifying areas for improvement, implementing corrective actions, and monitoring the effectiveness of the improvements made.

Formal approaches to SQA:

Statistical Software Quality Assurance (SQA) and Six Sigma are two formal approaches to improving software quality that use statistical methods to identify and eliminate defects in the software development process.

- **Statistical SQA:** It involves collecting and analyzing data to identify trends, patterns, and anomalies in the software development process. It uses statistical tools and techniques to determine the process's capability and identify areas for improvement. It focuses on reducing variability, improving quality, and optimizing the software development process.
- **Six Sigma:** It is a methodology for process improvement that uses statistical methods to identify and eliminate defects in the software development process. It aims to achieve a level of performance that corresponds to only 3.4 defects per million opportunities. Six Sigma uses a structured approach, called DMAIC (Define, Measure, Analyze, Improve, and Control), to improve the process and eliminate defects.

SQA Plan:

A Software Quality Assurance (SQA) Plan is a document that outlines the quality objectives, processes, procedures, and standards for a software development project. The SQA Plan is typically developed by the Quality Assurance team in collaboration with the project team and stakeholders. It serves as a guide for the project team to ensure that the software is developed in accordance with the specified quality requirements and standards.

The SQA Plan typically includes the following elements:

- **Quality objectives:** The quality objectives are the overall goals for the software development project. They define the quality characteristics that the software should possess, such as reliability, efficiency, usability, and maintainability.

- **Roles and responsibilities:** The SQA Plan should define the roles and responsibilities of the Quality Assurance team, project team, and stakeholders. It should also outline the communication channels and reporting mechanisms.
- **Quality standards:** The SQA Plan should define the quality standards that the software should adhere to, such as ISO standards, industry best practices, or company-specific standards.
- **Testing and validation:** The SQA Plan should define the testing and validation activities that will be performed to ensure that the software meets the specified requirements and quality standards. It should also define the test methodologies, tools, and techniques that will be used.
- **Process improvement:** The SQA Plan should define the process improvement activities that will be performed to continuously improve the software development process. It should also define the metrics and data collection mechanisms that will be used to measure the process's performance.
- **Documentation:** The SQA Plan should define the documentation requirements for the software development project, including requirements documents, design documents, testing documents, and user manuals.
- **Change management:** The SQA Plan should define the change management process that will be used to manage changes to the software development process, requirements, design, and code.

Formal Technical Review:

Formal Technical Review is a structured process for reviewing a software product or artifact to identify defects, errors, or potential improvements. FTR is a collaborative process that involves a group of individuals with diverse backgrounds, skills, and perspectives.

FTR is a powerful tool for identifying defects and improving the quality of software products. It allows for collaboration and knowledge sharing between team members and can be used throughout the software development lifecycle, from requirements gathering to code review. By using FTR, software developers can ensure that their software products are of high quality, reliable, and meet the end-user's requirements.

Objectives of FTR:

The objectives of Formal Technical Reviews (FTR) are as follows:

- **Defect detection:** One of the primary objectives of FTR is to detect defects in the software product or artifact being reviewed. The review process is designed to identify issues and problems that can affect the software's functionality, usability, and maintainability.
- **Improvement of software quality:** FTR helps to improve the quality of the software product by identifying and eliminating defects, errors, and potential improvements. The review process ensures that the software is developed according to industry standards, best practices, and customer requirements.

- **Knowledge sharing:** FTR provides an opportunity for knowledge sharing and collaboration among team members. The review process allows team members to share their expertise, insights, and feedback on the software product or artifact being reviewed.
- **Cost reduction:** FTR can help to reduce the cost of software development by identifying and eliminating defects early in the development process. This reduces the need for costly rework and ensures that the software is developed on time and within budget.
- **Risk mitigation:** FTR can help to mitigate the risk associated with software development by identifying defects and potential improvements early in the development process. This ensures that the software is developed to meet the customer's requirements and reduces the risk of project failure.

Guidelines for conducting FTR:

The following are the general guidelines to conduct a Formal Technical Review (FTR) effectively:

- **Planning:** The FTR process should be well-planned in advance, including the selection of the software product or artifact to be reviewed, the participants, and the review objectives. It is important to set a clear and specific agenda for the review meeting.
- **Preparation:** Participants should prepare for the review meeting by studying the software product or artifact, identifying potential issues, and preparing comments or questions. This includes reviewing the requirements, design, code, and any related documentation.
- **Review meeting:** The review meeting should be conducted in a structured and organized manner. The software developer should present the software product or artifact to the review team, and the team should then discuss the software product or artifact, identify defects or potential improvements, and generate a list of action items.
- **Follow-up:** After the review meeting, the software developer should address the identified defects or potential improvements and implement the action items generated by the review team. A follow-up review meeting can be conducted to ensure that the defects have been addressed.
- **Review team:** The review team should consist of individuals with diverse backgrounds, skills, and perspectives. The team should include subject matter experts, developers, quality assurance personnel, and other stakeholders.
- **Objectives and criteria:** The objectives and review criteria should be clearly defined in advance. The objectives should be focused on improving software quality, and the review criteria should be based on industry standards, best practices, and customer requirements.
- **Recording and reporting:** The FTR process should be well-documented, including the review meeting minutes, defect reports, and action items. This documentation should be distributed to all relevant stakeholders and used to track the progress of the review process.

ISO Standards:

ISO (International Organization for Standardization) is an independent, non-governmental international organization that develops and publishes standards for various industries and sectors. ISO develops and publishes a wide range of standards that cover different aspects of products, services, and systems. These standards are developed through a consensus-based process and are used by organizations to ensure quality, safety, and efficiency in their operations.

ISO has developed several standards related to software quality. Some of the widely used ISO standards in software quality include:

- **ISO/IEC 12207:** This is a standard for software life cycle processes. It provides a framework for the development, operation, and maintenance of software systems.
- **ISO/IEC 15504:** This is a standard for process assessment, also known as the Software Process Improvement and Capability Determination (SPICE) standard. It provides a framework for assessing the maturity of software processes and improving them.
- **ISO/IEC 9126:** This is a standard for software product quality. It provides a framework for measuring the quality characteristics of software products, including functionality, reliability, usability, efficiency, maintainability, and portability.
- **ISO/IEC 25010:** This is a standard for software product quality, also known as the System and software Quality Models standard. It provides a framework for evaluating the quality of software products, including the quality of the software itself, as well as the quality of the software's documentation and support materials.
- **ISO/IEC 27001:** This is a standard for information security management systems. It provides a framework for managing and protecting sensitive information in software systems.
- **ISO/IEC 29119:** This is a standard for software testing. It provides a framework for planning, designing, executing, and reporting software tests.

ISO 9000 and its family:

ISO 9000 is a family of international standards for quality management systems. These standards provide a framework for organizations to establish, implement, maintain, and continuously improve their quality management systems. While the ISO 9000 family is not specifically designed for software development, it is widely applied in the software industry.

ISO 9001 is the standard in the ISO 9000 family that defines the requirements for a quality management system. It provides a set of guidelines for organizations to follow in order to ensure that their products and services meet customer requirements and are consistently of high quality. The ISO 9001 standard includes requirements for management commitment, process control, documentation, measurement and analysis, and continuous improvement.

The ISO 9001 standard can be applied to software development in a variety of ways, including:

- **Process improvement:** ISO 9001 can help software development organizations to improve their processes, resulting in more efficient and effective development processes.
- **Customer satisfaction:** ISO 9001 requires organizations to focus on customer requirements and to measure customer satisfaction. This can help software development organizations to deliver products and services that meet customer needs and expectations.
- **Risk management:** ISO 9001 requires organizations to identify and manage risks, which can help software development organizations to identify and manage potential risks associated with software development projects.
- **Quality assurance:** ISO 9001 provides a framework for quality management, including requirements for process control, documentation, and measurement and analysis. This can help software development organizations to establish and maintain effective quality assurance practices.

Capability Maturity Model Integration:

Capability Maturity Model Integration (CMMI) is a process improvement framework that is used to assess and improve the effectiveness, efficiency, and quality of an organization's processes. It is a comprehensive set of guidelines that helps organizations to improve their performance by identifying areas of weakness and implementing best practices.

The CMMI framework is used by organizations across a wide range of industries, including software development, engineering, manufacturing, and service industries. The framework is organized into five levels of process maturity, ranging from initial to optimized, with each level building upon the previous one. The five levels are:

- **Initial:** Processes are ad hoc and often chaotic, with little or no documentation.
- **Managed:** Processes are defined and documented, and the organization has some level of control over them.
- **Defined:** Processes are well-defined, documented, and standardized across the organization.
- **Quantitatively Managed:** Processes are measured and controlled using metrics, and the organization has a data-driven approach to process improvement.
- **Optimized:** The organization has a continuous process improvement culture, and processes are continually improved and refined.

There are many examples of organizations that have successfully implemented the Capability Maturity Model Integration (CMMI) framework to improve their processes and performance. Here are a few examples:

- **Lockheed Martin:** One of the world's largest aerospace and defense companies, Lockheed Martin has used CMMI to improve its software development processes. By implementing CMMI, the company has reduced defects in its software, improved delivery times, and increased customer satisfaction.

- **IBM:** IBM has used CMMI to improve its software development and service delivery processes. The company has achieved CMMI Level 5 certification, the highest level of process maturity, which has helped it to increase productivity, reduce costs, and deliver high-quality products and services.
- **Tata Consultancy Services (TCS):** TCS, a global IT services and consulting company, has used CMMI to improve its software development and delivery processes. By implementing CMMI, TCS has been able to improve the quality of its software, reduce costs, and increase customer satisfaction.
- **Raytheon:** Raytheon, a major defense contractor, has used CMMI to improve its processes for developing complex systems and software. By implementing CMMI, the company has improved the quality and reliability of its systems, reduced development time, and increased customer satisfaction.
- **Siemens:** Siemens, a multinational conglomerate, has used CMMI to improve its product development processes. By implementing CMMI, the company has been able to standardize its processes across different business units, improve product quality, and reduce costs.

Software Configuration Management:

Software Configuration Management (SCM) is a process that involves identifying, organizing, controlling, and tracking changes to software, as well as documentation and other related items, throughout the software development lifecycle. SCM is a critical component of software engineering and is used to manage the software development process, ensure quality, and facilitate collaboration among team members.

SCM involves the use of tools, techniques, and processes to manage and control changes to software and related artifacts. This includes managing source code, documentation, testing artifacts, and other items related to the software development process. SCM also involves maintaining a history of changes made to the software, tracking the status of items, and ensuring that only authorized changes are made.

Some key aspects of software configuration management include version control, change management, and build management. Version control involves tracking changes to source code and other artifacts over time, while change management involves ensuring that changes are made according to a defined process, are properly tested, and do not cause unintended consequences. Build management involves building and packaging the software for deployment, as well as managing dependencies and ensuring that the correct versions of all components are included.

By implementing effective SCM processes, software development teams can reduce errors, improve collaboration, and ensure that software is delivered on time, within budget, and with the required level of quality.

Objectives of SCM:

The main objectives of Software Configuration Management (SCM) are:

- ***Version control:*** One of the primary objectives of SCM is to ensure that all software artifacts are versioned, tracked, and controlled. Version control ensures that the correct versions of source code, documentation, and other software artifacts are used throughout the software development lifecycle.
- ***Change management:*** Another important objective of SCM is to manage and control changes to software artifacts. Change management involves ensuring that changes are made according to a defined process, are properly tested, and do not cause unintended consequences.
- ***Collaboration:*** SCM also aims to facilitate collaboration among team members by providing tools and processes for managing and sharing software artifacts. This includes providing a centralized repository for source code, documentation, and other artifacts, as well as tools for reviewing and merging changes made by different team members.
- ***Quality control:*** SCM plays a crucial role in ensuring software quality. By implementing rigorous change control processes, testing procedures, and quality assurance checks, SCM helps to identify and fix defects early in the development process, reducing the risk of introducing errors into the software.
- ***Build and release management:*** SCM also helps to manage the build and release process, ensuring that the software is built correctly, packaged, and delivered on time and to the appropriate stakeholders. This includes managing dependencies, automating builds, and providing tools for testing and deployment.

Baselines in SCM:

In Software Configuration Management (SCM), a baseline refers to a snapshot of the state of the software and related artifacts at a specific point in time. A baseline is a reference point against which changes can be tracked and compared. It is used to establish a point of reference for future development, testing, and release activities.

A baseline typically includes a set of related software artifacts, such as source code, documentation, and configuration files. The baseline may also include other related items such as test plans, requirements, and user manuals. Baselines can be created at different stages of the software development lifecycle, such as after the completion of a major milestone or after a significant change has been made to the software.

Once a baseline has been established, changes to the software and related artifacts can be tracked and compared against the baseline. This allows developers and other stakeholders to monitor changes, identify problems, and ensure that changes are made according to a defined process. Baselines also provide a mechanism for rolling back changes if problems are encountered during the development or testing process.

Layers in SCM:

The layers of Software Configuration Management (SCM) are:

- **Identification:** This layer involves identifying and documenting all software artifacts that are to be controlled and managed by the SCM process. This includes identifying the source code, documentation, configuration files, test scripts, and other artifacts that are part of the software development process. The identification layer is important to ensure that all the software artifacts are known and controlled throughout the software development lifecycle.
- **Version control:** This layer involves managing and controlling the different versions of software artifacts. Version control helps in tracking changes to software artifacts and ensuring that the correct versions of the artifacts are used throughout the software development process. The version control layer includes creating new versions, labeling versions, and managing multiple versions of the same artifact.
- **Change control:** This layer involves managing and controlling changes to software artifacts. Change control helps in ensuring that changes are made according to a defined process, are properly tested, and do not cause unintended consequences. The change control layer includes defining a process for submitting and reviewing change requests, testing and verifying changes, and approving or rejecting changes.
- **Configuration auditing and status reporting:** This layer involves auditing and reporting on the status of software artifacts. This includes monitoring the changes made to software artifacts, ensuring that they are properly tested, and documenting any changes made to the artifacts. Configuration auditing and status reporting are important to ensure that the software development process is well-managed, transparent, and repeatable.

CASE tools for Configuration Management:

Computer-Aided Software Engineering (CASE) tools can be used to assist with different aspects of Software Configuration Management (SCM), including version control, change management, and release management. Here are some examples of CASE tools that can be used for SCM:

- **Git:** Git is a distributed version control system that is widely used in software development. It allows developers to track changes to source code and other software artifacts and manage different versions of the artifacts.
- **Subversion:** Subversion (also known as SVN) is a centralized version control system that allows developers to manage and track changes to software artifacts.
- **Perforce:** Perforce is a commercial version control system that is used in software development to manage and track changes to software artifacts.
- **JIRA:** JIRA is a project management tool that can be used for change management in software development. It allows developers to submit, track, and manage change requests throughout the software development process.

- ***Jenkins:*** Jenkins is a continuous integration and deployment tool that can be used for release management. It allows developers to automate the build and deployment process, ensuring that software releases are properly tested and deployed to production.
- ***Ansible:*** Ansible is a configuration management tool that can be used to manage the configuration of software environments. It allows developers to define and manage the configuration of servers, databases, and other infrastructure components.

8.4 OBJECT-ORIENTED FUNDAMENTALS AND ANALYSIS

Defining Models:

- **Algorithmic modeling**: It refers to a procedural programming approach where the program is developed by breaking down the task into a set of sequential steps, each of which is designed to perform a specific function. The emphasis is on the logic of the program, and the code is organized around the algorithms. In algorithmic modeling, the focus is on the process, rather than the data.
- **Object Oriented modeling:** It is an approach to software development that emphasizes the use of objects, which are instances of classes, to represent real-world entities. The focus is on the data and the relationships between objects. The emphasis is on the objects, their behavior, and their interactions, rather than the algorithms that manipulate them.

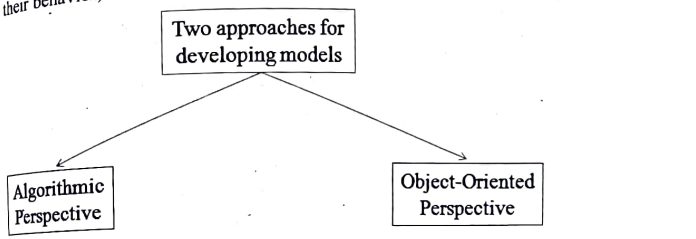


Figure: Classification of approaches for developing models.

Object Model:

In software engineering, an object model is a conceptual representation of the data, behavior, and relationships of a software system. It defines the objects, their attributes, methods, and relationships that exist within the system. The object model can be divided into two types of elements: major elements and minor elements.

Major elements are:

- **OO Abstraction:** Abstraction is the process of simplifying complex systems by focusing on their essential characteristics and ignoring irrelevant details. In OO programming, abstraction is achieved through the use of classes, which describe the common attributes and behaviors of objects of a certain type. By creating abstract classes that define the essential characteristics of objects, programmers can simplify complex systems and make them easier to understand and modify.
- **OO Encapsulation:** Encapsulation is the process of hiding the implementation details of objects from the rest of the program. In OO programming, encapsulation is achieved through the use of access modifiers, such as public, private, and protected. By hiding the implementation details of objects, programmers can reduce complexity, increase security, and make the program easier to maintain.
- **OO Hierarchy:** Hierarchy is the process of organizing objects into a tree-like structure based on their relationships. In OO programming, hierarchy is achieved through the use of inheritance, which allows objects to inherit properties and behaviors from other objects. By organizing objects into a hierarchy, programmers can create more complex and flexible systems, with the ability to add or override properties and behaviors as needed.
- **OO Modularity:** Modularity is the process of dividing a software system into smaller, more manageable parts. In OO programming, modularity is achieved through the use of objects, which are self-contained units of code that encapsulate data and behavior. By dividing a software system into smaller, more manageable objects, programmers can increase maintainability, reduce complexity, and improve code reuse.

Minor elements are:

- **OO Typing:** Typing refers to the process of defining the data types of objects and their attributes. In an OO system, each object has a specific type that defines its behavior and properties. The type of an object is defined by its class, which specifies the methods and attributes that are available to instances of the class. The typing system in an OO system provides strong type checking, which helps to catch errors at compile time rather than at runtime.
- **OO Concurrency:** Concurrency refers to the ability of an OO system to handle multiple tasks or processes simultaneously. In a concurrent system, multiple threads or processes can access and modify shared resources, which can lead to data races and other synchronization problems. To manage concurrency, OO systems typically use techniques such as locks, semaphores, and atomic operations to ensure that shared resources are accessed in a safe and predictable manner.
- **OO Persistence:** Persistence refers to the ability of an OO system to store data and state across different sessions or instances of the program. In a persistent system, data can be stored in a database, file, or other external storage medium, and retrieved later when needed. To achieve persistence, OO systems typically use techniques such as object serialization and deserialization, which convert objects and their data into a format that can be stored and retrieved from a persistent storage medium.

Object Oriented Analysis:

- Understands the problem.

Object Oriented Design:

- Understands the solution.

OO Analysis vs OO Design:

Object-oriented (OO) analysis and OO design are two important phases of the software development life cycle. While they share some similarities, they differ in terms of their objectives and outcomes. Here are some key differences between OO analysis and OO design:

Object-Oriented Analysis:

- **Objective:** OO analysis is focused on understanding the problem domain and identifying the requirements of the system.
- **Outcome:** The outcome of OO analysis is a clear and comprehensive description of the problem domain and the requirements of the system.
- **Activities:** The key activities in OO analysis include identifying objects and their relationships, defining use cases and scenarios, and creating a conceptual model of the problem domain.
- **Deliverables:** The deliverables of OO analysis include use case diagrams, activity diagrams, class diagrams, and a conceptual model of the problem domain.
- **Emphasis:** OO analysis emphasizes the problem domain and the requirements of the system.

Object-Oriented Design:

- **Objective:** OO design is focused on designing the solution for the requirements identified during OO analysis.
- **Outcome:** The outcome of OO design is a detailed and complete design of the system, including the architecture, components, and interactions.
- **Activities:** The key activities in OO design include defining the system architecture, creating detailed design specifications, and creating implementation plans.
- **Deliverables:** The deliverables of OO design include sequence diagrams, collaboration diagrams, class diagrams, and implementation plans.
- **Emphasis:** OO design emphasizes the solution domain and the design of the system.

Overall, OO analysis and OO design are both important phases of the software development life cycle, and they work together to ensure that software systems are designed and developed to meet the requirements of the problem domain. While they have different objectives and outcomes, they are both critical for the successful development of high-quality software systems.

Requirement process:

The requirement process in software engineering is the set of activities that are undertaken to gather, analyze, document, validate, and manage the requirements of a software system. Requirements are the foundation of any software system, and the requirement process is critical to ensure that the software system meets the needs of its users.

The requirement process typically involves the following activities:

- **Requirements gathering:** This involves identifying stakeholders and gathering their needs and expectations for the system. Techniques such as interviews, surveys, and workshops can be used to gather requirements.
- **Requirements analysis:** This involves analyzing the requirements to ensure that they are clear, complete, and consistent. This is an iterative process that may involve refining and prioritizing the requirements.
- **Requirements documentation:** This involves documenting the requirements in a clear and concise manner. The documentation should include the functional and non-functional requirements of the system.
- **Requirements validation:** This involves ensuring that the requirements are correct and meet the needs of the stakeholders. This can be done through techniques such as prototyping, simulations, and reviews.
- **Requirements management:** This involves tracking and maintaining the requirements throughout the software development process. Changes to the requirements should be tracked and managed to ensure that they do not impact the overall system.

Object oriented development cycle:

The Object-Oriented (OO) development cycle, also known as the Object-Oriented Software Development (OOSD) cycle, is a software development process that uses an object-oriented approach for designing, developing, and maintaining software systems.

The OO development cycle typically includes the following stages:

- **Requirements gathering and analysis:** This stage involves identifying the requirements of the software system and analyzing them to ensure that they are complete, consistent, and correct. Use cases, user stories, and other requirement analysis techniques can be used to identify the functional and non-functional requirements of the system.
- **Design:** In this stage, the software system is designed based on the requirements identified in the previous stage. The design should be based on object-oriented principles and should include class diagrams, sequence diagrams, and other UML diagrams to represent the structure and behavior of the system.
- **Implementation:** During this stage, the software system is implemented based on the design. The implementation should follow the principles of object-oriented programming (OOP) and should use the appropriate programming language and development tools.

- **Testing:** In this stage, the software system is tested to ensure that it meets the requirements and is free of defects. Testing can include unit testing, integration testing, system testing, and acceptance testing.
- **Maintenance:** Once the software system is deployed, it needs to be maintained to ensure that it continues to meet the needs of its users. Maintenance can involve bug fixes, updates, and enhancements.

The OO development cycle is an iterative process, and the stages can be revisited as needed. For example, if defects are found during testing, the implementation stage may need to be revisited to fix the defects. Similarly, if requirements change during maintenance, the requirements analysis and design stages may need to be revisited to incorporate the changes. By following the OO development cycle, software developers can ensure that software systems are designed and developed based on object-oriented principles and that they meet the requirements of their users.

Use Cases:

Use cases are a way of describing the interactions between actors and a system, and how the system responds to those interactions. A use case is a scenario that describes a specific interaction between a user or actor and a system, and is typically written in natural language or a visual diagramming language like UML.

Use case naming examples:

- Purchase CAN-InfoTech Ticket ..... Excellent
- Purchase CAN-InfoTech Tickets ..... Very Good
- Purchase Ticket (Not much details) ..... Good
- Ticket purchase (passive) ..... Fair
- Ticket Order (System view, not user) ..... poor
- Pay for ticket (Procedure, not process) ..... Not Acceptable

Types of use cases:

- **High-level use case (Brief):** A high-level use case, also known as a summary use case, provides an overview of the system's functionality from a high-level perspective. It typically describes the main goals or objectives of the system and the major features or functions that are required to achieve those goals. High-level use cases are useful for communicating the overall vision and scope of the system to stakeholders.
- **Expanded use case (Fully Dressed):** An expanded use case, also known as a detailed use case, provides a more detailed description of a particular feature or function of the system. It typically includes a step-by-step description of the interactions between actors and the system, along with any preconditions, postconditions, and alternate paths that may be required. Expanded use cases are useful for documenting the specific requirements of the system in more detail.

- **Essential use case (Black box):** An essential use case, also known as a business use case, describes the fundamental requirements of the system from the perspective of the user or customer. It focuses on the external behavior of the system, without specifying its implementation details. Essential use cases are useful for capturing the high-level functional requirements of the system and ensuring that the system meets the needs of its users.
- **Concrete use case (White box):** A concrete use case, also known as a scenario use case, provides a specific example of how the system is used in a particular context. It typically includes a description of the actors involved, the steps they take to interact with the system, and the expected results. Concrete use cases are useful for testing and validating the system's functionality, as well as for providing examples of how the system is used in practice.

Guidelines for creating use cases:

- **Identify the actors:** Identify the different types of users or external systems that will interact with the system, and define their roles and responsibilities.
- **Define the goal:** Define the main objective or goal of the use case, and ensure that it aligns with the overall goals of the system and the stakeholders' requirements.
- **Use clear and concise language:** Use clear, concise, and unambiguous language to describe the interactions between the actors and the system, and avoid technical jargon or complex terminology.
- **Identify the preconditions:** Identify any conditions or constraints that must be met before the use case can be executed, such as user authentication or system availability.
- **Describe the main flow of events:** Describe the sequence of actions that the actor takes to achieve the goal of the use case, and ensure that it includes all the steps required to complete the task.
- **Include alternative flows:** Identify any alternative paths or exceptions that may occur during the execution of the use case, such as error conditions or unexpected inputs.
- **Define the postconditions:** Define the expected results or outcomes of the use case, such as updated data or confirmation messages.
- **Keep the use case focused:** Keep the use case focused on a single task or objective, and avoid including unnecessary details or functionality.
- **Validate the use case:** Validate the use case with the stakeholders and other members of the development team to ensure that it accurately represents the system's functionality and meets the stakeholders' requirements.

Unified Modeling Language:

- UML provides a set of graphical notations for modeling software systems, including classes, objects, use cases, activities, interactions, and state machines.
- UML is used to describe the structure, behavior, and interactions of a software system.

- UML is not a programming language, but rather a modeling language that is used to design and describe software systems before they are implemented.

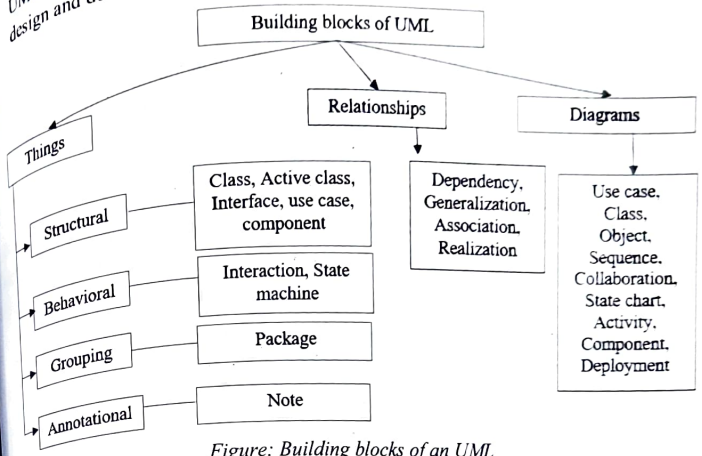


Figure: Building blocks of an UML

Things in UML:

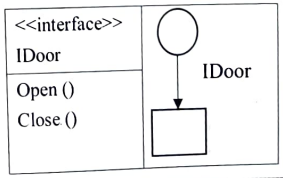
- **Structural Things:** These are the elements that represent the static structure of a software system, including classes, objects, components, and packages.
  - **Class:** A class is a template that defines the properties and methods of a set of objects. It is the blueprint for creating instances of the objects.

    Notation:

Door
Origin Size
Open () Close ()

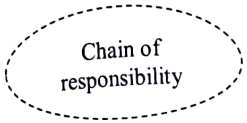
  - **Interface:** An interface defines a contract or a set of rules that must be followed by any class that implements it. It specifies the signature of methods that the class must provide, but it does not provide any implementation details.

    Notation:

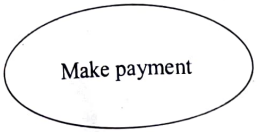
- **Collaboration:** A collaboration represents a group of objects that work together to achieve a common goal. It is a way of modeling the interactions between objects in a system.

  Notation:



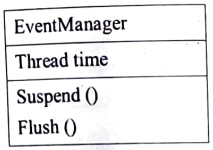
- **Use case:** A use case represents a specific functionality that the system must provide to its users. It describes the behavior of the system from the perspective of the user.

  Notation:



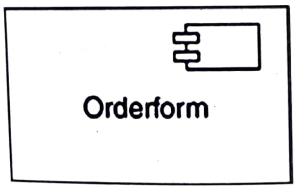
- **Active class:** An active class represents an object that is capable of initiating and handling its own events. It is used to model objects that have a significant amount of internal behavior.

  Notation:



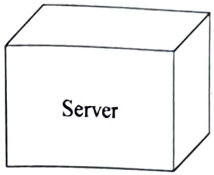
- **Component:** A component represents a modular and reusable part of a system that can be easily replaced or updated. It can be either physical or software-based.

  Notation:



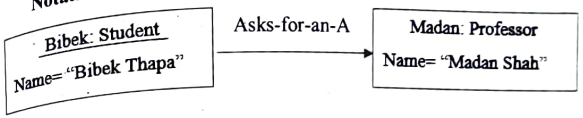
- **Node:** A node represents a physical resource in the system, such as a server, a computer, or a database. It is used to model the deployment of the system and the distribution of its components across different resources.

Notation:



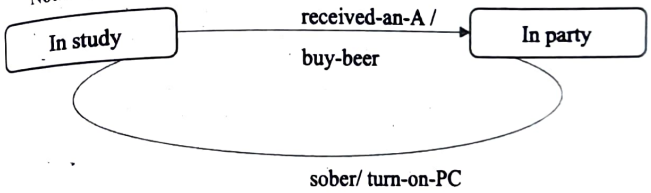
- **Behavioral Things:** These are the elements that represent the dynamic behavior of a software system, including use cases, interactions, state machines, and activities.
  - **Interaction:** One object exchanges message with another object to perform specific tasks.

    Notation:



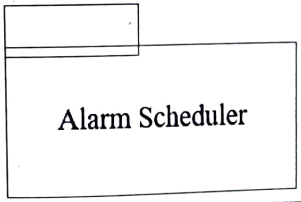
  - **State Machine:** Some object may go lifetime to multiple sequence of states.

    Notation:



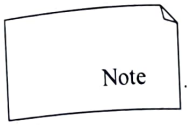
- **Grouping Things:** These are the elements that are used to group and organize other elements, including packages, subsystems, and nodes.
  - **Packages:** Organizing the elements into a group.

    Notation:



- **Notes:** Used to render comments and constraints.

Notation:

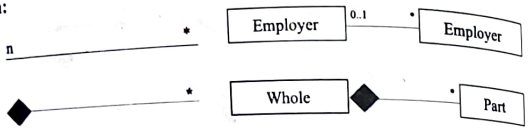


- **Annotational Things:** These are the elements that are used to add additional information and comments to a UML diagram, including notes and stereotypes.

Relationships in UML:

- **Association:** An association represents a relationship between two or more classes. It describes how the classes are connected or related to each other. Associations can be one-to-one, one-to-many, or many-to-many.

Notation:



- **Aggregation:** "Has a" relationship.

Notation:

Even if the container is destroyed then its content is not destroyed.

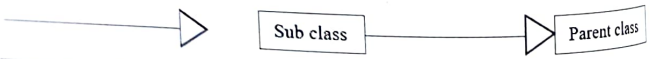
- **Composition:** "Owns a" relationship. A composition is a relationship between two classes where one class is made up of one or more instances of the other class.

Notation:

If container is destroyed then its content is also destroyed.

- **Generalization:** A generalization is a relationship between two classes where one class is a specialized version of the other. The specialized class (called the subclass or derived class) inherits the attributes and behaviors of the more general class (called the superclass or base class) and may add its own unique features.

Notation:



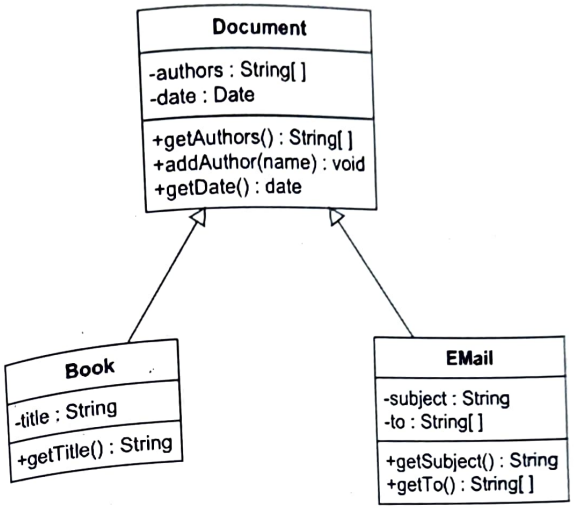


Figure: Generalization

- **Realization:** Connecting 2 elements in which one element describes responsibility but does not implement whereas another element implements.

Notation:

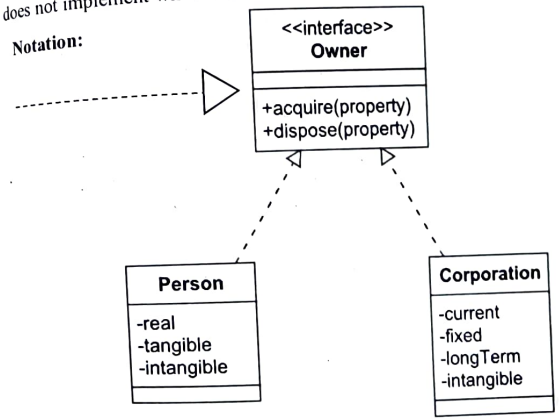
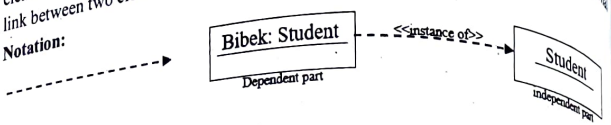


Figure: Realization

- **Dependency:** A dependency is a relationship between two elements where a change to one element (the independent element) may affect the other element (the dependent

element). It is a weak relationship that represents a temporary, situational or directional link between two elements.

Notation:



UML Diagrams:

- **Use case diagram:** Use case diagrams are used to model the interactions between actors (users or external systems) and a system to achieve specific goals. They show the high-level functional requirements of a system, and the various ways in which actors can interact with it to achieve their goals. Use case diagrams consist of use cases (represented as ovals) and actors (represented as stick figures). Use cases are descriptions of the steps an actor takes to achieve a particular goal. Actors are external entities that interact with the system, and can be users or other systems. Types of actors:
  - **Primary actor:** User who access the system to obtain his goals. E.g., cashier
  - **Supporting actor:** This actor provides services to the system. E.g.: automated payment authorization system for bank
  - **Offstage actor:** Shows interest in the system but has no any contribution. E.g., Government Tax agency.
- **Class diagram:** shows the classes, interfaces, and their relationships to each other in a system.
- **Object diagram:** shows a specific instance of a class and the relationships between its objects.
- **Sequence diagram:** illustrates the interactions between objects in a system over time.
- **Collaboration diagram:** similar to a sequence diagram, but emphasizes the interactions between objects.
- **State machine diagram:** shows the behavior of a system or class in response to external events.
- **Activity diagram:** represents the flow of actions in a system or process, including decision points and branching paths.
- **Component diagram:** depicts the components of a system and their relationships.
- **Deployment diagram:** shows how a system is deployed in a physical or virtual environment.
- **Package diagram:** organizes the elements of a system into packages and their dependencies.
- **Composite structure diagram:** shows the internal structure of a class or component.
- **Timing diagram:** illustrates the behavior of a system in response to time-based events.
- **Interaction overview diagram:** provides an overview of the interactions between objects and their relationships.

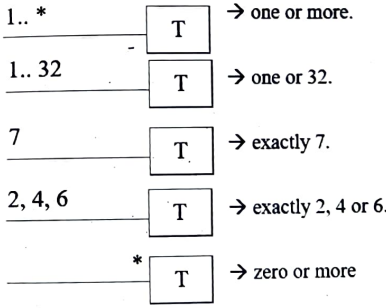
Conceptual class modeling:

Conceptual class modeling, also known as domain modeling, is the process of creating a conceptual model of the entities, attributes, and relationships in a problem domain. It is a technique used in software engineering to capture and represent the essential features of a problem domain, which is the area of knowledge or activity that a software system will address.

The main purpose of conceptual class modeling is to create a high-level, abstract representation of the problem domain that can be used to design and develop the software system. The model consists of classes, attributes, and relationships, and serves as a basis for creating more detailed and concrete models, such as the object model and the database schema.

Steps in creating domain model:

- Find conceptual classes
  - By reuse or modifying existing models.
  - Use category list.
  - Identifying noun phrases.
- Draw domain model
- Add Associations:
  - **Associations:** Relationship between 2 or more conceptual class
  - **Role:** End of association is role. Role may have:
    - **Multiplicity Expression:** How many instances of class A can be associated with one instance of class B.



    - **Naming Associations:** Should start with capital letter. E.g., TypeName_VerbPhrase_TypeName. **"Paid_by" or "PaidBy"**
    - **Navigation:** There is a link between two objects, A link contains direction along with message called the navigation. Message can flow in forward or backward direction.

- Add Attributes:

Representation of System Behavior:

System Behavior: Representation of what a system does? Representation can be done using

- Use case diagram
- Interaction diagram (System sequence diagram, collaboration diagram)
- Operation contract

Operation contract:

We used use case to describe the system behavior in Unified Processing (UP) which is sufficient. But sometimes we require more detailed description of a system behavior, for this reason we need operation for contract. Operation Contracts are described in terms of preconditions and postconditions.

Contract CO2: enterItem

Operation:	enterItem(itemID: ItemID, quantity: integer)
Cross References:	Use Cases: Process Sale
Preconditions:	There is a sale underway.
Postconditions:	– A SalesLineItem instance sli was created *(instance creation).* – sli was associated with the current Sale *(association formed).* – sli.quantity became quantity *(attribute modification).* – sli was associated with a ProductDescription, based on itemID match *(association formed).*

Figure: An example of Operation Contract

8.5 OBJECT-ORIENTED DESIGN

OO Analysis to OO Design:

OOA (Object-Oriented Analysis) and OOD (Object-Oriented Design) are two phases in the software development process that are closely related but serve different purposes.

OOA is the process of analyzing a problem domain to identify the objects, their attributes, relationships, and behaviors. The goal of OOA is to identify the objects and their characteristics that are relevant to the problem at hand.

OOD, on the other hand, is the process of designing a solution to the problem identified during OOA. The goal of OOD is to create a solution that satisfies the requirements identified during OOA by designing a set of classes and their associated behaviors.

In order to convert from OOA to OOD, you would typically follow these steps:

- **Review the OOA model:** Review the analysis model created during OOA to identify the objects, attributes, relationships, and behaviors.

- **Identify candidate classes:** Based on the OOA model, identify the classes that are required to solve the problem.
- **Identify attributes and methods:** For each candidate class, identify the attributes and methods required to represent the class and its behavior.
- **Create a class diagram:** Use a class diagram to represent the classes and their relationships. A class diagram shows the classes, their attributes, and the relationships between the classes.
- **Refine the design:** Refine the design by adding more details to the classes and their behaviors.
- **Review and iterate:** Review the design and iterate if necessary. The design should be reviewed to ensure that it meets the requirements identified during OOA.
- **Implement the design:** Once the design is complete, the implementation can begin.

Overall, the process of converting from OOA to OOD is an iterative process that involves refining the design until it meets the requirements identified during OOA. It's important to review the design and iterate if necessary to ensure that the final solution is effective and efficient.

Describing and Elaborating Use Cases:

Follow these steps in developing the use case model:

- **Developing an initial use case**
  - Identify and develop the problem statement.
  - Determine the major actors who uses the system and corresponding use cases of the system.
  - Create an initial rough use case diagram.
  - Describe in detail the use cases.
  - Perform textual analysis to identify or refine the domain class.
- **Refine the Use case**
  - Base use case description needs to be developed.
  - Now the base use case description needs to be iteratively elaborated and determine the <<extend>>, <<include>> and generalization relationship.
  - Instance scenario needs to be developed.
  - Prioritize the use case.

<<include>> and <<extend>> in use case:

Include Relationship: The include relationship is used when one use case includes another use case. The included use case is a necessary step in the execution of the including use case. The include relationship is shown using a solid arrow from the including use case to the included use case, labeled with the keyword "include".

Extend Relationship: The extend relationship is used when one use case is optional and can be extended by another use case. The extending use case adds functionality to the extended use case. The extend relationship is shown using a dashed arrow from the extending use case to the extended use case, labeled with the keyword "extend".

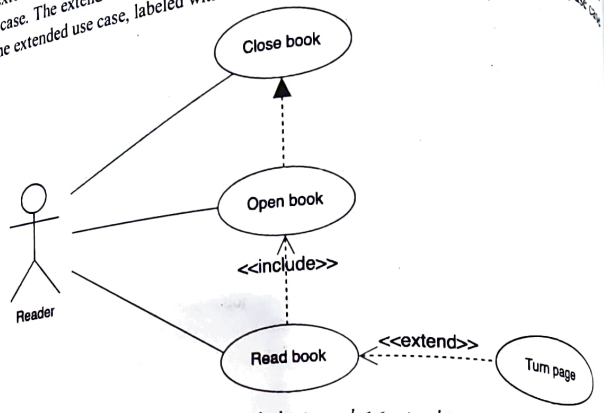


Figure: Utilization of <<include>> and <<extends>> relationship

In above use case diagram,

- "Read book" use case includes "Open book" use case. This means a reader must open book to read a book. It is mandatory.
- "Read book" use case extends "Turn page" use case. This means a reader can still read a book without turning a page. Its Optional.

Collaboration Diagram:

A collaboration diagram in UML (Unified Modeling Language) is a type of interaction diagram that shows how objects collaborate or interact to achieve a particular goal or behavior. Collaboration diagrams are also known as communication diagrams. A collaboration diagram consists of a set of objects (or actors) and the messages that are exchanged between them to accomplish a task. The objects are represented as rectangles, and the messages are represented as arrows connecting the objects. The arrows show the flow of messages between the objects. In a collaboration diagram, the focus is on the interactions between the objects and the order in which the messages are sent and received. The sequence of the messages is shown by numbering the arrows.

To create a collaboration diagram in UML, follow these steps:

- Identify the actors and objects involved in the interaction. This may involve reviewing a use case or scenario to determine which objects and actors are required.
- Create a rectangle for each object and actor in the interaction.

- Draw arrows between the objects to represent the messages that are exchanged. The arrows should be labeled to indicate the message being sent.
- Number the arrows to show the sequence of the messages.
- Add any necessary conditions or constraints to the diagram, such as loops or alternative paths.
- Review and refine the diagram as necessary to ensure that it accurately represents the interaction between the objects.

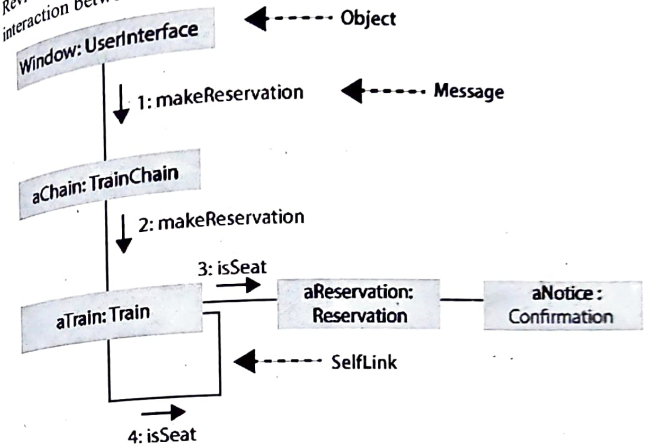


Figure: An example of collaboration diagram

Patterns and Design Patterns:

Patterns: A reusable piece of the software that provides some kind of functionality.

Design patterns: During a software development phase, a developer may face many problems. Design pattern is the solution to those problems. Categories of a design patterns:

- **General Responsibility Assignment Software Pattern (GRASP)**
- **Gang of four (GoF)**

GRASP Patterns classifications:

- **Creator →Who creates?**

  Problem: Who creates an object A?

  Solution: Assign class B the responsibility to create object A if one of these is true (more is better)

  - B contains or compositely aggregates A
  - B recordsA
  - B closely uses A
  - B has the initializing data for A

- **Information Expert → Who, in the general case, is responsible?**

  Problem: What is a basic principle by which to assign responsibilities to objects?

  Solution: Assign a responsibility to the class that has the information needed to fulfill it.

- **Low Coupling → Support low dependency and increased reuse**

  Problem: How to reduce the impact of change? How to support low dependency, and increased reuse?

  Solution: Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

- **Controller → Who handles a system event?**

  Problem: What first object beyond the UI layer receives and coordinates "controls" a system operation?

  Solution: Assign the responsibility to an object representing one of these choices:

  - Represents the overall "system", "root object", device that the software is running within, or a major subsystem (these are all variations of a facade controller)

- **High Cohesion → How to keep complexity manageable?**

  Problem: How to keep objects focused, understandable, manageable and as a side effect support Low Coupling?

  Solution: Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

- **Polymorphism → Who, when behavior varies by type?**

  Problem: How to handle alternatives based on type?

  Solution: When related alternatives or behaviors vary by type (class), assign the responsibility for the behavior (using polymorphism operations) to the types for which the behavior varies.

- **Pure Fabrication → Who, when you are desperate, and do not want to violate High Cohesion and Low Coupling?**

  Problem: What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, but solutions offered by other principles are not appropriate?

  Solution: Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept.

- **Indirection → Who, to avoid direct coupling?**

  Problem: Where to assign a responsibility to avoid direct coupling between two or more things?

  Solution: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

- **Protected Variations (Don't talk to strangers) → Who, to avoid knowing about the structure of indirect objects?**

  Problem: How to design objects, subsystems and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?

  Solution: Identify points of predicted variation or instability, assign responsibilities to create a stable interface around them.

GoF patterns categories:

1. Creational Design Patterns (Abstract factory, builder, factory method, prototype, singleton)
2. Structural Design Patterns (Adaptor, bridge, composite, decorator, façade, flyweight, proxy)
3. Behavior Design Patterns (Chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method, visitor)

The Gang of Four (GoF) design patterns are commonly categorized into three groups:

- **Creational Patterns:** These patterns provide a way to create objects while hiding the creation logic from the client. The main focus of creational patterns is to create objects in a manner that is suitable for the situation. The five GoF creational patterns are:
  - **Singleton Pattern:** This pattern ensures that only one instance of a class is created and provides a global point of access to that instance.
  - **Factory Method Pattern:** This pattern provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
  - **Abstract Factory Pattern:** This pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.
  - **Builder Pattern:** This pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
  - **Prototype Pattern:** This pattern creates new objects by cloning existing ones, rather than creating new objects from scratch.
- **Structural Patterns:** These patterns deal with the composition of classes and objects to form larger structures. They provide a way to simplify the relationships between objects by defining ways in which they can interact with one another. The seven GoF structural patterns are:
  - **Adapter Pattern:** This pattern converts the interface of a class into another interface that clients expect.
  - **Bridge Pattern:** This pattern decouples an abstraction from its implementation so that the two can vary independently.
  - **Composite Pattern:** This pattern allows clients to treat individual objects and groups of objects uniformly, by representing them as a single object.

- **Decorator Pattern:** This pattern adds new behaviors to an object dynamically by wrapping it with a decorator object.
- **Facade Pattern:** This pattern provides a simple interface to a complex subsystem by grouping related functionality into a single class.
- **Flyweight Pattern:** This pattern reduces the memory usage of large numbers of similar objects, by sharing common parts between them.
- **Proxy Pattern:** This pattern provides a placeholder for another object, to control access to it.

- **Behavioral Patterns:** These patterns deal with the communication and interaction between objects. They focus on the responsibilities and behaviors of objects, rather than their structure. The eleven GoF behavioral patterns are:
  - **Chain of Responsibility Pattern:** This pattern creates a chain of objects to handle a request, allowing multiple objects to handle the request without the client knowing which one will handle it.
  - **Command Pattern:** This pattern encapsulates a request as an object, allowing clients to parameterize objects with different requests and to queue or log requests.
  - **Interpreter Pattern:** This pattern defines a language and its grammar, and then uses an interpreter to interpret sentences in the language.
  - **Iterator Pattern:** This pattern provides a way to access the elements of an object sequentially without exposing its underlying representation.
  - **Mediator Pattern:** This pattern defines an object that encapsulates how a set of objects interact, allowing them to be decoupled from each other.
  - **Memento Pattern:** This pattern allows an object to capture its internal state and save it externally, so that the object can be restored to that state later.
  - **Observer Pattern:** This pattern defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.
  - **State Pattern:** This pattern allows an object to alter its behavior when its internal state changes, by using different classes to represent the different states.
  - **Strategy Pattern:** This pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable at runtime.
  - **Template Method Pattern:** This pattern defines the skeleton of an algorithm in a superclass, but lets subclasses override specific steps of the algorithm without changing its structure.
  - **Visitor Pattern:** This pattern defines a new operation to be performed on the elements of an object structure, without changing the classes of the elements.

Some Developer's problems solved by GoF patterns:

- ***Singleton:***
- ***Problem:*** How do you ensure that it is never possible to create more than one instance of a singleton class?
- ***Observer:***
- ***Problem:*** How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems?
- ***Delegation:***
- ***Problem:*** How can you most effectively make use of a method that already exists in the other class?
- ***Adapter:***
- ***Problem:*** How to obtain the power of polymorphism when reusing a class whose methods have the same function but not the same signature as the other methods in the hierarchy?
- ***Façade:***
- ***Problem:*** How do you simplify the view that programmers have of a complex package?

Determining Visibility:

Visibility: Ability of one object to:

- See another object or,
- Have reference to another object

Visibility from "A" to "B" can occur in 4 ways

- **Attribute Visibility:** "B" is an attribute of "A".
- **Parameter Visibility:** "B" is parameter of method "A".
- **Local Visibility:** "B" is local object in method "A".
- **Global Visibility:** "B" is in some way globally visible.

Design Class Diagrams (DCD):

Design class diagram provides the detail of the class and methods associated with that class.

Steps in creating DCD:

- First, determine the software classes and represent those class by class diagram.
- Add method names.
- Add type information (types of attributes, parameters of methods and return values).
- Add associations and navigability.
- Add dependency (indicated with dashed arrow lines ·····►)

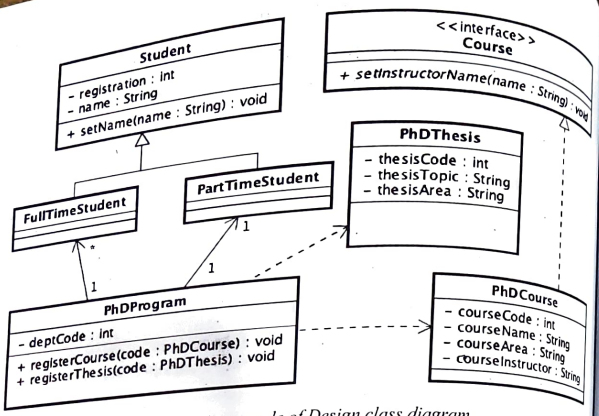


Figure: An example of Design class diagram

8.6 OBJECT-ORIENTED DESIGN IMPLEMENTATION

Mapping designs to code

The interaction diagram and the class diagrams can be used as input to the code generation process. The implementation model is a model which consists of several implementation artifacts such as source code, database definition, HTML pages and so on.

Various object-oriented languages such as java, c++, c#, small talk, python and so on can be used as the languages of implementation. Following are the approaches used for generating code from the design.

Steps for mapping design to codes

1. CREATING CLASS DEFINITIONS FROM DCDS
2. CREATING METHODS FROM INTERACTION DIAGRAMS
3. CONTAINER/COLLECTION CLASSES IN CODE
4. ORDER OF IMPLEMENTATION
5. EXCEPTIONS AND ERROR HANDLING

MULTIPLE CHOICE QUESTIONS

1. **What are the important characteristics of a good software?**
   A. Software is developed or engineered; it is not manufactured in the classical sense.
   B. Software doesn't "wear out".
   C. Software can be custom built or custom build.
   D. All mentioned above
2. **Compilers, Editors software come under which type of software?**
   A. System software
   B. Application software
   C. Scientific software
   D. Bespoke software
3. **Software Engineering is defined as systematic, disciplined and quantifiable approach for the development, operation and maintenance of software.**
   A. True
   B. False
4. **What is the full form of RAD Software process model?**
   A. Rapid Application Development.
   B. Relative Application Design and Development.
   C. Rapid Application Design.
   D. Recent Application Development.
5. **In software engineering, Software project management *SPM (contains of a number of activities, which contains _____.**
   A. Project planning
   B. Scope management
   C. Project estimation
   D. All mentioned above
6. **Which of the following option is not defined in a good Software Requirement Specification (SRS) document?**
   A. Functional Requirement.
   B. Nonfunctional Requirement.
   C. Goals of implementation.
   D. Algorithm for software implementation.
7. **Which of the following is the simplest model of software development life cycle?**
   A. Spiral model
   B. Agile model
   C. Incremental model
   D. Waterfall model
8. **Which of the following is the understanding of software product limitations, learning system related problems or changes to be done in existing systems beforehand, identifying and addressing the impact of project on organization and personnel etc.?**
   A. Software Specification
   B. Feasibility Study
   C. Requirement Elicitation
   D. System Analysis
9. **Which design identifies the software as a system with many components interacting with each other?**
   A. Architectural design
   B. Low-level design
   C. Blueprint design
   D. Both B & C