**Software Engineering**

**Software**

Software is a set/collection of instructions(programs) that when executed provide desired features or functionality.

**Software comprises of:**

* **Instructions:** Actual code/programs
* **Data structure:** Enables the software to organize and manipulate data
* **Documentation:** guides for usage and maintenance.

**Types**

**Generic/off the shelf:** made for use by many users.

**Custom Software/besopke:** Developed for a specific client or purpose.

**Engineering**

Engineering is the application of mathematical and scientific principles to develop and maintain, structure, software, that solve real world problems.

**IEEE definition**

Software engineering is application of systematic, quantifiable, disciplined approach to the development, maintenance and operation of software.

**Objectives of Software Engineering**

1. **Correctness:** The software must perform the desired task correctly without errors.
2. **Cost-Effectiveness:** The software must be budget friendly and worth the cost.
3. **Scalability:** Software must be able to handle large amounts of data/users.
4. **Reusability:** Software must be made in such manner that its code is reusable to build other software.
5. **Maintainability:** Software must be easy to fix and maintain.
6. **Efficient:** Software must use allocated resources efficiently
7. **Reliable:** Software must be able to perform its task without failure

**Need for Software Engineering**

Software engineering is essential as modern codebases are larger and complex to handle. Software engineering provides a systematic and disciplined approach to design, develop, test and maintain a software efficiently. Without software engineering a software can face issues such as software crashes, missed deadlines and budget overrun.

Learning software engineering dev learns about:

1. Managing complexity: larger software are complex and hard to handel.
2. Improving quality: ensures that software are manageable and reliable.
3. Cost & time effectiveness: Reduces development time and cost.
4. Team collaboration: Manages large team and codebases.
5. Systematic approach: Gives structured methods (like SDLC).
6. Avoid System failure: Avoids software crisis

**Software Crisis**

Software crisis refers to early days of software development (1960s – 1980s) when software became larger and complex to handle. Developers struggled to deliver quality software within deadline and assigned budget.

**Major Issues**

1. **Budget Overrun:** Projects kept exceeding the budget due to poor planning and estimation
2. **User dissatisfaction:** There was a significant difference between were the user’s need and what the final product delivered.
3. **Poor quality:** Poor code quality resulted in underperformance.
4. **Lack of structure methods:** No fixed process or planning was followed.
5. **Difficult maintenance:** Due to lack of structure or documentation software once deployed were really hard to fix and maintain.
6. **Late delivery:** Due to poor planning projects kept getting late.

Software engineering crisis didn’t get solved instantly it was addressed gradually with the help of better tools, management skills and structured approach.

**Key Solutions:**

1. **Introduction of Software Engineering:** Provided systematic and disciplined approach to development.
2. **Software Development Life Cycle (SDLC):** Different model like waterfall, spiral etc brought structure to software development.
3. **Better project management techniques:** Better time management, resource allocation and risk management.
4. **Use of design principle and patterns:** Made code more modular, reusable and maintainable.
5. **Improved testing:** using more modern tools to thoroughly test software allowed to deliver more refined project.

**\*\*TO DO\*\***

**MYTH**

* **Myths by Managers:**
* **Myths by Developers:**
* **Myths by Users:**

|  |  |
| --- | --- |
| **Program** | **Software** |
| Program is a set of instructions written in a specific programming language | Software is a collection of programs and documentation |
| Program have very limited functionality | A software can have a lot of functionality such as GUI, IO, processing etc. |
| Program are smaller in size | Software can be huge in size |
| Programs may be unplanned, unorganized and unstructured | Software must be planned, organized and structured |
| Programs can have a patchy mostly technical documentation | Software must have a thorough detailed documentation with user manuals |
| Dev of program takes less time | Dev of software can take a lot of time depending upon the scale. |

**Brooks’ No Silver Bullet**

There is no single magical solution (no silver bullet) that can instantly solve all the problems in software development and dramatically improve productivity, reliability, or simplicity.

**Key Points:**

* Software development is inherently complex.
* No tool or method can provide a 10x improvement overnight.
* Most challenges are essential (built into the nature of software), not accidental (just tooling issues).

**Essential vs Accidental:**

* Essential complexity = Problem-solving, understanding user needs.
* Accidental complexity = Tools, syntax, or language-specific issues.

**ISO 9126 Software Quality Model**

ISO 9126 is a software quality evaluating model which comprise of 6 main quality

characteristics:

1. **Functionality:** Ability to perform the task software was made to do.
   1. Correctness: Are the functions provided by software right set of functions.
   2. Accuracy: Is the software accurate.
   3. Security: is the software safe from attacks?
2. **Usability:** How pleasant and easy is the software to use:
   1. Can a user easily understand and use the software.
   2. How long does it take for a new user to learn.
   3. Is it easy to operate in daily task
3. **Reliability:** Can the software perform its functions for a set time
   1. How many times the software fails.
   2. Does the software still works after error is generated.
   3. How long does the software takes to recover from errors.
4. **Maintainability:** Is the software easy to maintain and modify.
   1. Can you find issues or bugs easily.
   2. Is it easy to make changes in the software.
   3. Does new updates introduce new bugs.
5. **Portability:** Can the software run on different platforms.
   1. Is the software able to run on different platforms and OS.
   2. Is the installation process easy
6. **Efficiency:** how efficient is the software
   1. How much times is taken by the software to produces the results.
   2. How the software is managing the allocated resources.

**SDLC (Software Development Life Cycle)**

SDLC is a systematic and cost-effective structured process followed to develop high quality software.

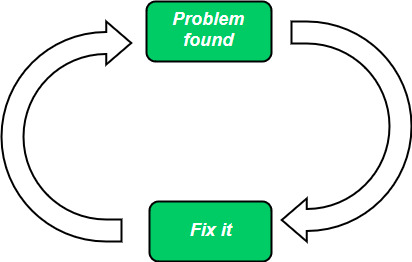
**Phases**

1. Planning (requirement analysis): understand what client/user needs.
2. System Design: Plan the system architecture (UI, Database, etc.)
3. Implementation: Actual code based on design.
4. Testing: check for bugs and make sure software works as expected.
5. Deployment: Deliver the final product.
6. Maintenance: Fix any bugs, issues or update features.

**SDLC Models**

**Code & Fix Model**

It is the most basic and oldest software development model where developers start coding immediately without any former planning or design and handle the errors or issues that arise on the go.



**Phases**

1. **Code:** Start writing code as soon as idea is known.
2. **Fix:** Fix bugs and make any changes that are required.

**Advantages**

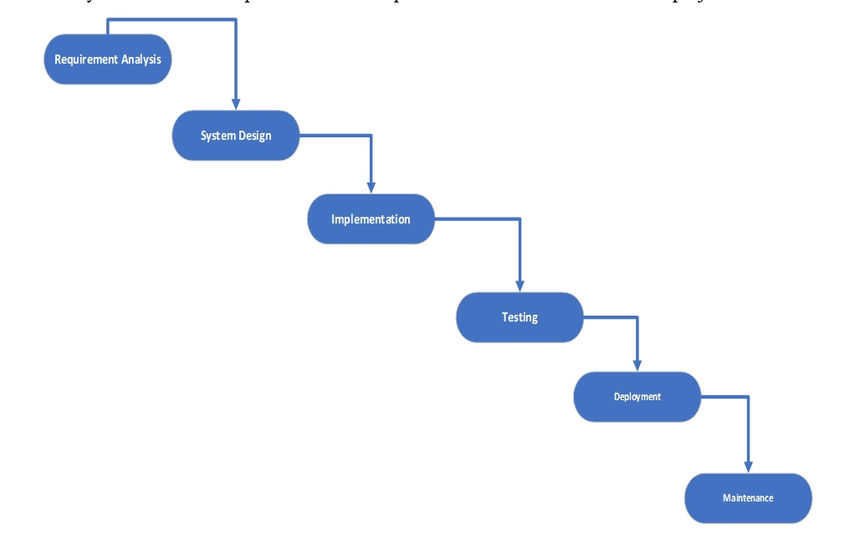
* Fast for smaller projects.
* Good for new developers.
* Easy to start.
* Low budget.

**Disadvantages**

* No structure of the project.
* Hard to maintain.
* Not scalable.

**Waterfall model**

Waterfall model is like a linear and sequential software development model where each phase must be followed in sequence and next phase can only be started after the current one is completed, you cannot go back to any phase that’s been completed already.



**Phases**

1. **Requirement analysis:** Gather and document information, what user needs.
2. **System Design:** Plan the architecture of the system such as UI, database etc.
3. **Implement:** Write the code according to the system design.
4. **Testing:** Test the code to make sure it works as expected.
5. **Deployment:** Deliver the working project to user.
6. **Maintenance:** Fix any bugs, issues or update features.

**Advantages**

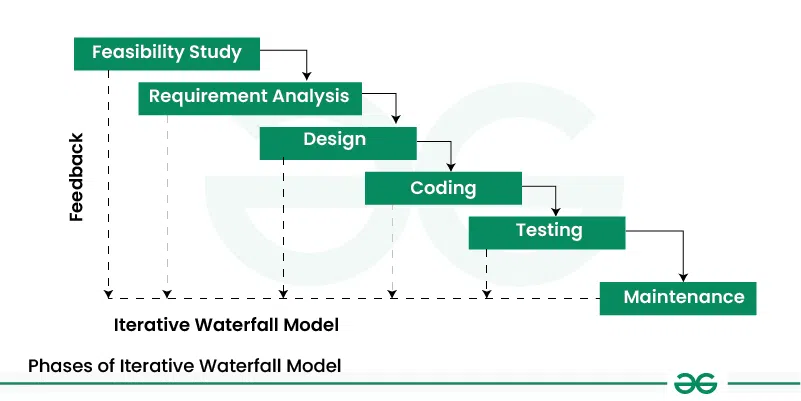
* Simple and easy to understand.
* Good for small and well-understood projects.
* Phases are well defined.
* Phases processed one at a time.
* Project management is easy.
* All phases are well documented.

**Disadvantages**

* Very rigid no overlap permissions (no going back).
* Late testing (bugs and issues are found very late in project dev).
* Risky for large complex projects
* Not suitable for continually changing requirement projects.

**Iterative waterfall model**

Iterative Waterfall is a modified version of the traditional Waterfall model that allows feedback and repetition between phases. You can go back to a previous phase if needed — unlike the strict linear model.



**Phases**

* 1. **Feasibility Study(Can’t go back to this)**
     + Technical feasibility: Can we build it?
     + Economic feasibility: Is it cost-effective?
     + Operational feasibility: Will users accept it?
     + Legal feasibility: Any legal constraints?
  2. **REST SAME AS WATERFALL**

**Advantages:**

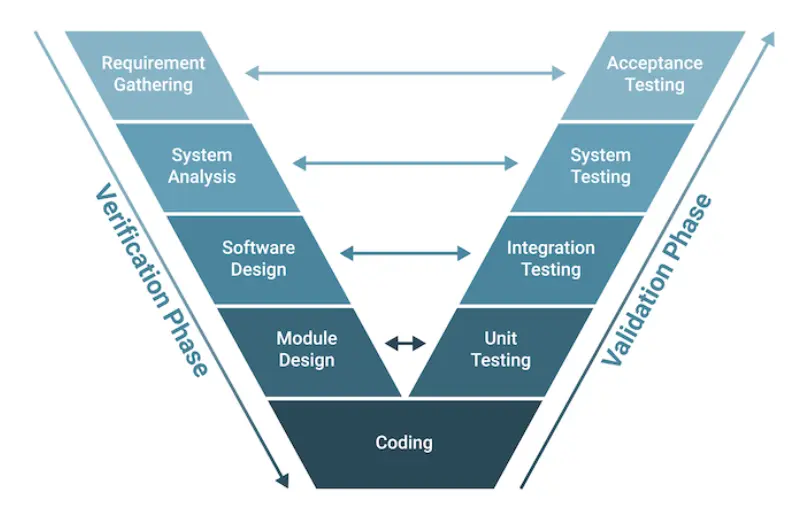
* More flexible than traditional Waterfall due to feedback loops
* Early detection and correction of errors
* Better handling of minor requirement changes
* Clear, structured phases with documentation
* Easier to manage compared to completely agile models

**Disadvantages:**

* Still rigid as it’s hard to accommodate major changes
* Not suitable for highly dynamic or uncertain projects
* Rework can increase cost and time
* Feedback needs good and clear communication to be effective

**V-Model**

V-model is a modified version of the waterfall model where each development phase is linked to a testing phase. It is shaped like a V dev on the left and testing on the right.



**Phases**

1. **Requirement Analysis:** Gather information about what user wants
   1. Acceptance Testing ensures final product meets user needs.
2. **System Design:** Plan the architecture of the components and how they interact.
   1. System Testing checks the whole system’s functionality.
3. **Architecture Design (high level design):** Break down components into multiple modules and define interface.
   1. Integration testing check module interactions
4. **Module Design (Low level design):** Define logic for each module.
   1. Unit Testing checks each module independently.
5. **Coding:** Actual development based on above designs.

**Advantages**

* Testing is planned in parallel with development.
* Bugs can be found in early stages.
* Suitable for small projects with clear requirements.

**Disadvantages**

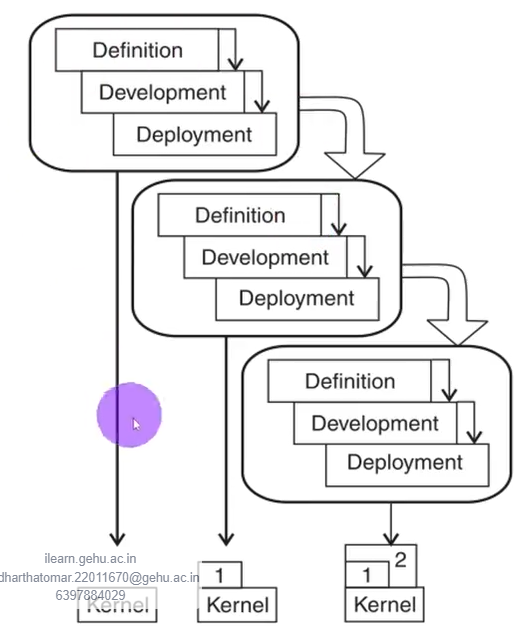
* No flexibility for changing requirements.
* Not suitable for complex or evolving projects.
* Rigid like waterfall.

**Incremental Implementation**

In incremental implementation we divide the software into parts called increments. Each increment is then developed, tested and deployed individually gradually building the whole system.

**Phases**

1. **Requirement Analysis:** What the user wants.
2. **System Design (core features):** Plan the architecture of the system such as UI, database etc.
3. **Development & deploy (core features):** Develop, test and deploy.
4. **Design, Develop & deploy (next increment):** Again design, develop, test and deploy in form of updates.
5. **Repeat the process until whole software is developed**

****

**Advantages**

* Reduces initial delivery time.
* Easier to test, debug and deploy smaller modules.
* Flexibility to change features according to the feedback.

**Disadvantages**

* Requires proper planning and design to divide the project in smaller parts.
* Integration issues may arise if interface doesn’t communicate properly.
* Could be costlier from a single-phase development.
* All increment should be part of design.

**Prototyping model**

Prototyping is a process of creating a simpler version of the software to get feedbacks from the user or customer, to better understand the requirements of the user. Then we refine the prototype to make it full-scale software.

**Types:**

**Throwaway Prototyping**

Throwaway prototyping also known as rapid prototyping is the process of creating a prototype as fast as possible to gather the feedbacks. The prototype is then discarded and the real software development starts from scratch.

**Phases**

1. Quick Requirement gathering
2. Building prototype (simple, quick)
3. User feedback and Evaluation
4. Discard prototype
5. Development of the final system

**Advantages**

* Helps clarify requirements early
* Clears any misunderstanding between customer and developer
* Saves time by identifying problems early

**Disadvantages**

* Extra cost and effort as prototype is discarded
* Prototype could be overly simplified
* User might misunderstand prototype as final product

**Evolutionary Prototype**

Evolutionary prototype is an iterative method where the initial prototype is continuously refined based on user feedbacks and made into the final product.

**Phases:**

1. Initial Requirement analysis
2. Build initial prototype
3. User Eval and feedback
4. Refine prototype (add features, fix issues, etc)
5. Repeat 3,4,5 until final system is ready

**Advantages**

* Flexible for changing requirements
* User involved throughout the development
* No work wasted as prototype becomes the final product

**Disadvantages**

* Can extend development time due to excess iterations
* Requires strong user commitment for multiple feedbacks
* Documentation can be weak due to focus on prototype

**Spiral Model**

Spiral model combines the iterative model with risk analysis, allowing to repeat the cycles (spirals) of planning, risk analysis, engineering and evaluation. It is designed to handle complex and high-risk projects.



**Phases**

**Planning**: Gather requirements and plan objective

**Risk analysis:** Identify and evaluate risks, plan mitigation

**Engineering:** Develop and test product increment

**Evaluation:** Review progress with user and plan next spiral

**Advantages**

* Focus on risk management to reduce project failures
* Early prototypes improve user feedback
* Suitable for large, complex and high-risk projects

**Disadvantages**

* Can be costly and time consuming due to repeated cycles
* Requires expertise in risk analysis
* Not ideal for small or low-risk projects.

**RAD (Rapid Application Development)**

RAD is an iterative and adaptive software dev model that focuses on rapid prototyping, reusable components and continuous user involvement to deliver system faster and with high quality.

**Phases**

1. **Business modelling:** First understand business goals, information flow and processes of an organisation. Understand how the information is being used.
2. **Data modelling:** Analyse data models and define how they interact with other data.
3. **Process modelling:** Define the logic for data models. Define how data is processed and transformed into outputs.
4. **Application generation:** Application and actual code is generated using automated tools and reusable components.
5. **Testing and deployment:** Test each and every component of the software. The entire application is validated by users. Bugs are fixed and once the software is ready it is deployed.

**Advantages**

* Reduces development time
* Increases re-usability of components
* Encourages customer feedback

**Disadvantages**

* Requires highly skilled developer and designer
* Heavily depends on tools available
* Inapplicable for cheaper projects as cost of modelling and automated code generation is very high

**RUP (Rational Unified Process)**

RUP is a software development model that provides, an iterative, disciplined and use-case driven approach. It clearly defines the roles, activities and deliverables and focuses on building high quality software incrementally.

**Phases**

1. **Inception:** Define business case and basic requirement, understand the project scope and goals.
2. **Elaboration:** Create architecture, refine requirement and address high risk elements
3. **Construction:** System design, program and testing. Develop a full system in iterations, integrate components.
4. **Transition:** Deploy system to users, do final testing, training and bug fixes.

**Advantages**

* **Iterative:** Develop in small cycles improving features
* **Use-case driven:** Focus on real-world usage scenarios
* **Architect centric:** Stay focus on design and architecture

**Disadvantages**

* Reduces risk early
* High quality output
* Adapts to changing

**Agile development**

Agile development is a software development framework that focuses on code rather than design, iterative approach, customer collaboration, flexible to changes.

**Key Principles (from Agile Manifesto):**

* Individuals and interactions over processes and tools
* Working software over documentation
* Customer collaboration over contract negotiation
* Responding to change over following a plan

**Advantages:**

* Faster and more flexible delivery
* High customer satisfaction
* Early detection of issues
* Better team collaboration

**Disadvantages:**

* Not ideal for fixed-scope or fixed-budget projects
* Needs skilled teams and active user participation
* Can lack documentation

**Unit 2**

**Requirements**

Requirements are the constraints or the capability that a software must follow or provide in order to meet user needs or business goals. They define how a software must look and behave.

**Two Types:**

1. **User Requirements**: These are high level requirements which are defined in natural language and are given from end point user’s POV. E.g. User should be able to login in g-mail.

 **Enduring Requirements:** Rarely change; based on domain or policy.  
E.g., "Only authorized users can access patient records."

 **Volatile Requirements:** Change over time due to business needs or tech upgrades.  
E.g., UI design preferences, third-party API usage.

1. **System Requirements:** Detailed technical description for developers. Describe system’s functions, constraints and operations.

It is of two types.

* 1. **Functional Requirements:**
     1. The basic features that a software must provide to the end user.
     2. They are necessary and must be added to the software as they are contracted to be in there.
     3. Each functionality must be defined clearly.
     4. Eg. User authentication on Login, verification e-mail to be sent on registeration
  2. **Non- Functional Requirements:**
     1. These are the quality constraints applied by the client.
     2. These can define what a UI must look like or the portability, maintainability, number of concurrent user’s software can handle, etc.
     3. These are extremely important and must be followed as failure might result in complete rejection of the project
     4. eg client expects 10000 credit card statement to be generated per min

**Requirement Engineering**

Requirement Engineering is the process of gathering, analysing, documenting and managing software requirements.

**Activities in Requirement Engineering**

1. **Feasibility Study:** This step checks the practicality and economical feasibility of the project. This includes checking the technical feasibility (do we have the tools and skill required), economic viability (can the project be completed within the budget) and any operational, legal or time bound-issues that could result in failure of project. This is done in order to save time and resources on projects which are too risky, costly or simply not achievable
2. **Requirement Gather (Elicitation):** This includes gathering all the specifications and features that are required to be in the software. This process includes meetings with stake holders, clients, users, brainstorming, interview, surveys etc. The goal is to have a clear understanding of what the user requires.
3. **Barrier in requirement Gathering:** Requirement gathering doesn’t always go smooth as expected there can be issues such as.
   1. **Communication Gap:** Developer and users using different jargons, this could cause misunderstanding.
   2. **Unclear Req:** The stakeholders are not sure of what they actually want, could lead to incomplete and vague inputs
   3. **Frequently changing req:** Req are being frequently changed, it gets hard to focus and build a proper system design.
   4. **Different expectation of stakeholder:** Every stakeholder has their own opinion and expectation, which may be hard to prioritize.
4. **Requirement Analysis:** After collecting every req, they need to be analysed for clarity, conciseness and conflicts. This process involves organising the req, removing duplicates, managing priority. This aims to create a blueprint that the developers can easily follow.
5. **Requirement Specification:** All the requirements are gathered in a place and properly formatted using a standard or software such as (SRS). This acts as a blueprint for the designed, developer and tester. This needs to be clear, unambiguous and with no conflicts. So that everyone involved should know what the software must do.
6. **Requirement Specification using Graph & Mathematic notation:** Sometimes just writing the req in a plain language is not enough. To clear any errors or misunderstanding between the developers and the stakeholders, graphs or mathematical expressions are used.

Graphs

In this notation various graphs, presentations or images such as ER diagrams, DFDs Use-case diagrams are used.

This ensures that developers properly understand the req and gives a clear idea to stake holders.

Mathematical Notation

Here various formulas, expressions and logical symbols are used to represent the project. This is mainly used in projects where correctness maters a lot eg banking or medical

1. **Requirement Validation:** Before development the final requirement document is checked for any errors or misunderstandings. This ensures that the final req are actually what the users wanted. This involves meeting with stakeholders, communication, prototyping, walkthroughs etc.
2. **Requirement Management:** Since req often change during dev process, this phase ensures that each change are being accommodated properly .it involves tracking, updating and controlling version control system properly and ensuring that, all the stakeholders are informed about every important change

**SRS (Software Requirement Specification)**

SRS is an important document, that contains all the requirements needed in the software. It acts as a contract between the developer and the client to ensure that every feature mentioned in the SRS is present in the software.

**Uses:**

* SRS contains all the requirements either functional or non-functional.
* It acts as blueprint for the designer, developer, tester and clients
* This also clears any misunderstanding between the developer and the stakeholders.
* System management engineers read the SRS document to know the structure, working and relation of the software with others.

**Good Practices:**

1. **Correctness:** A SRS must contain all the requirement details that are actually expected by the client from the system.
2. **Consistent:** SRS must have only one way of writing and it should not contradict any already set requirement. Eg switching from 12hrs format to 24hrs in a clock display.
3. **Concise:** SRS must have short and clear details.
4. **Unambiguous:** SRS requirements must only have 1 interpretation.
5. **Modifiable:** Must be able to accommodate changes to some extent.
6. **Testable:** Generate testcases and test plans from the documentation.
7. **Black-box view:** Must not have any design or implementational details

**Structure of an SRS Document (IEEE Standard 830):**

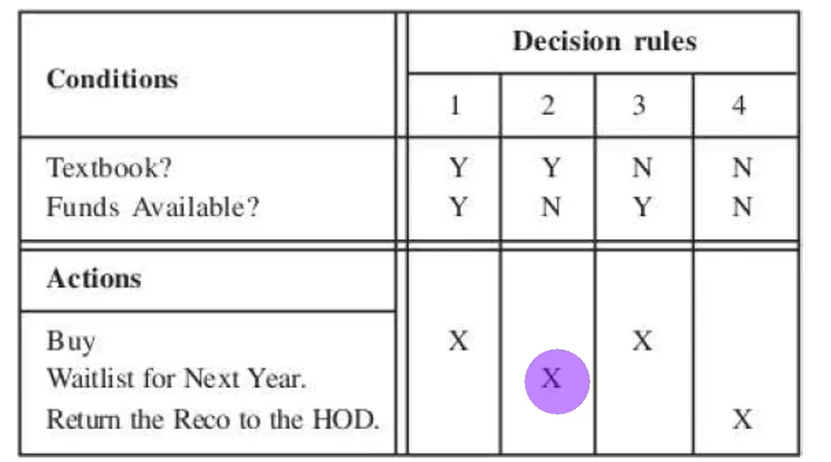
1. **Introduction**
   * Purpose, scope, definitions, and references.
2. **Overall Description**
   * Product perspective, functions, user needs, constraints.
3. **Specific Requirements**
   * Functional and non-functional requirements, interfaces, etc.
4. **Appendices & Glossary**
   * Any extra information or technical terms.

**Decision table**

A decision table is a method used to represent complex conditions and their corresponding actions. This is used mainly when there is a combination of two or more requirements that are affecting the decision.

**Advantages**

* Makes logic analysis easier
* Useful in requirement analysis and validation
* Helps in designing test cases

****

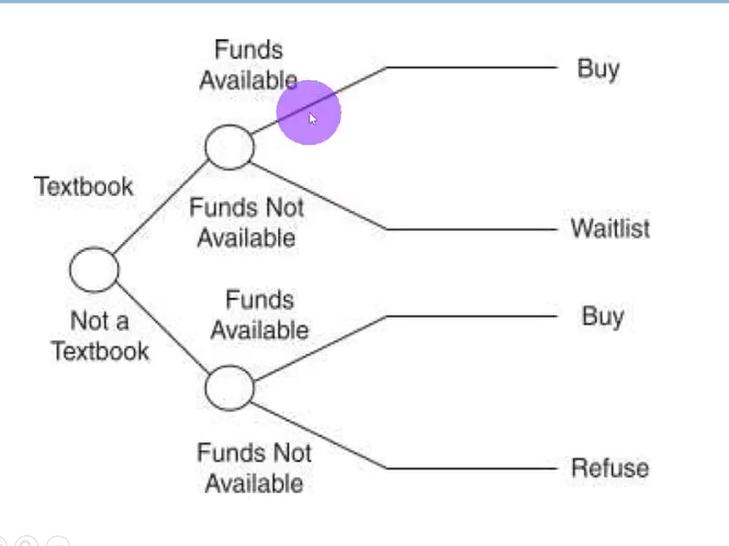
**Decision Tree**

A decision tree is a graphical representation of conditions and their corresponding result, these are very helpful for system which have a lot of options and rules.

A decision tree starts from its root nodes and branches.

Each node is a decision point where a choice has to be made.

Each branch has a corresponding value to the decision choice.



**Uses**

They can represent a large number of combinations of prior conditions with the result.

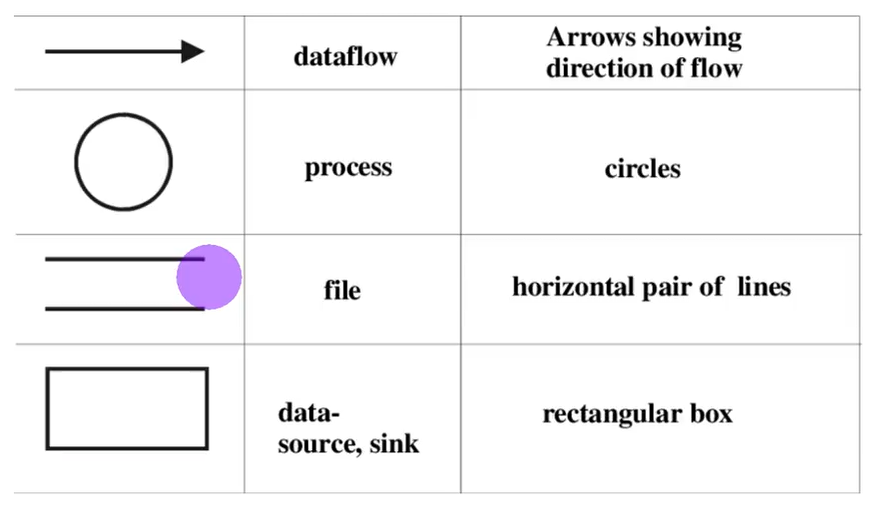
They are especially used for representation of order of decision making in the system

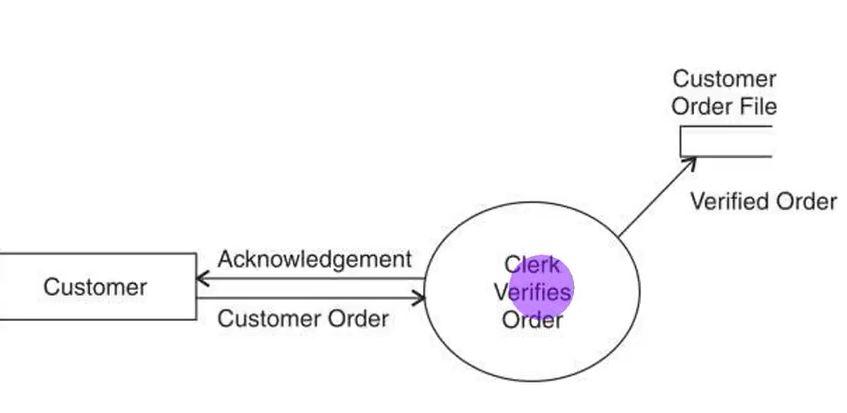
**DFD (Data Flow Diagram)**

DFD are a graphical representation of how data flows, is processed and is stored in the organisation’s system.

They are easy to understand by technical and non-technical people.

These are useful tools for database engineer, system programmer and analyst.





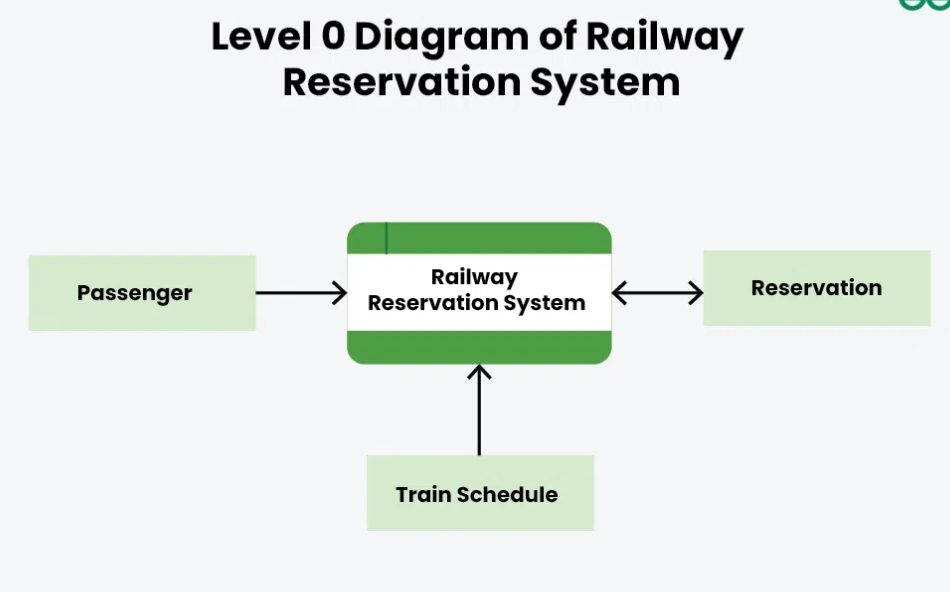
* The data flow depicts an input to or an output of a process. The data may be in any form such as a document record.

The **arrowhead** of the symbol indicates the direction of flow of data.

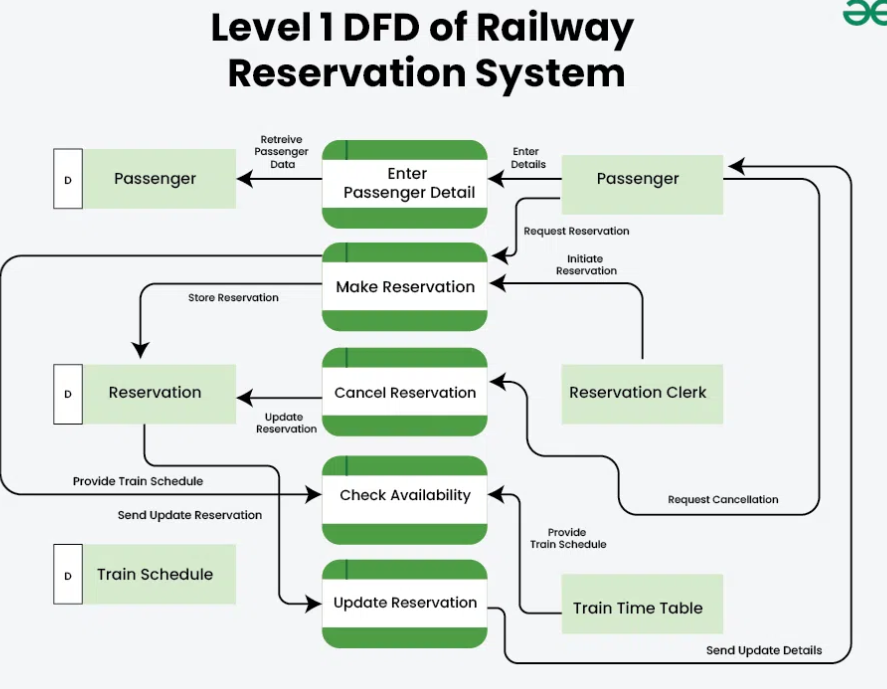
* **Circle** Performs operations like calculations, data validation, or transformations.
* Data store represents a storage location of data. (**Open Rectangle**)
* An external entity lies outside the boundary of the system under consideration. It may be the origin of certain data flow into the system boundary thus providing input to the system. (**Rectangle**).

**Types:**

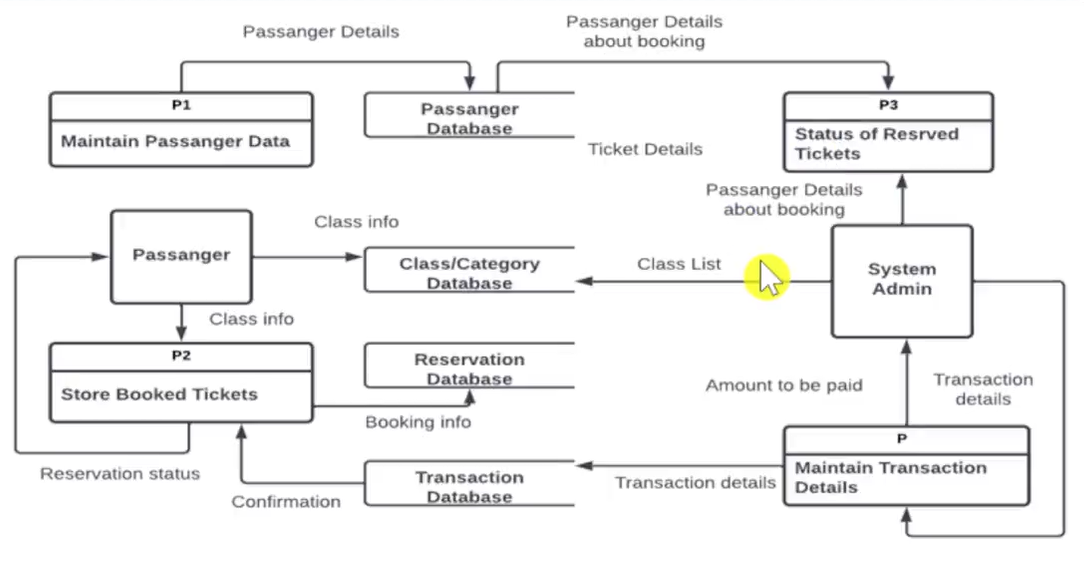
Level-0: Presents the overview of the system and its interactions with rest of world



Level-1: Presents more detailed version than Level 0 (context diagram) by showing sub-processes and stores of data.



Level-2: Certain elements of DFD are decomposed to more detailed model



**🔄 1. Finite State Machine (FSM)**

**✅ Definition:**

An FSM is a **mathematical model of computation** that represents a system using a **finite number of states**, **transitions**, and **inputs**.

**✅ Components:**

* **States**: Different situations of the system.
* **Transitions**: Movement from one state to another based on input.
* **Start State**, **Final State** (optional)

**✅ Use Case:**

* Used in **control systems**, **UI navigation**, **protocol design**, etc.

**✅ Example:**

Login screen:

* State 1: Awaiting input
* Input: Correct credentials → Transition
* State 2: Logged in

**✅ Advantage:**

* Simple and visual.
* Good for modeling **sequential** behavior.

**🔁 2. Statecharts (Harel State Diagrams)**

**✅ Definition:**

A **statechart** extends FSMs to handle **hierarchy**, **concurrency**, and **timing**—ideal for **real-time** and **event-driven** systems.

**✅ Features:**

* **Nested States**: States can contain substates.
* **Orthogonal Regions**: Model concurrent behavior.
* **Events/Actions**: Can trigger transitions or outputs.

**✅ Use Case:**

* Widely used in **UML diagrams**, **embedded systems**, and **control logic** for real-time systems.

**✅ Example:**

Microwave Oven:

* Door open → disable start
* Timer expires → switch to idle

**✅ Advantage:**

* More expressive than FSMs.
* Good for **complex systems** with multiple states and parallel actions.

**🔄 3. Petri Nets**

**✅ Definition:**

A **Petri Net** is a **graphical and mathematical modeling tool** for representing **concurrent**, **asynchronous**, and **distributed systems**.

**✅ Components:**

* **Places (circles)**: Represent conditions or states.
* **Transitions (rectangles)**: Events that change the state.
* **Tokens**: Represent the presence of resources or active conditions.
* **Arcs**: Connect places and transitions.

**✅ Behavior:**

When all input places to a transition contain tokens → transition fires → tokens are moved to output places.

**✅ Use Case:**

* Modeling **workflow**, **communication protocols**, **process scheduling**, **concurrency control**, etc.

**✅ Advantage:**

* Captures **parallelism**, **synchronization**, **resource sharing**.
* Formal analysis possible.

**Unit-3**

**Software Design**

Software design is the process of defining the structure, architecture, data, components and interface of a system software. This becomes the blueprint for the developers to follow while software development.

Goals

**1. Correctness**

* The design should correctly implement all the specified functional requirements.
* It must reflect what the software is expected to do.

**2. Efficiency**

* Optimizes resource usage such as CPU time, memory, and storage.
* Should avoid unnecessary computations and complexity.

**3. Maintainability**

* Easy to modify, enhance, and debug.
* Well-organized code and modularity promote maintainability.

**4. Reusability**

* Design should allow code and components to be reused in future systems.
* Promotes consistency and saves development effort.

**5. Modularity**

* Dividing the system into small, manageable, independent modules.
* Each module performs a single, well-defined function.

**6. Scalability**

* The system should be able to handle increased workload or growth easily.
* Design should support the addition of new modules or features.

**7. Security**

* Software should be resistant to unauthorized access or vulnerabilities.
* Design should include authentication, authorization, and encryption mechanisms.

**8. Robustness**

* The system should behave correctly even in unexpected or error conditions.

**9. Portability**

* Should be easy to adapt the software for use in different environments (OS, hardware).

Types

1. High Level Design (HLD): HLD defines the overall system architecture and module breakdown of the software. It shows how the system will be structured and meet the requirement of the SRS.
   1. Provides a blueprint for developers
   2. Defines to the stakeholders the flow of the system without using low-level terms.
   3. Identifies major modules and interfaces.
2. Low Level Desing (LLD): LLD also known as detailed design focuses in internal parts of the modules and the structure defined in the SRS. It contains the logic of each individual components and units
   1. This builds a developer friendly blueprint for coding, individual functions.
   2. It contains
   3. Pseudo code for functions
   4. Class & object diagram
   5. Error handling etc.

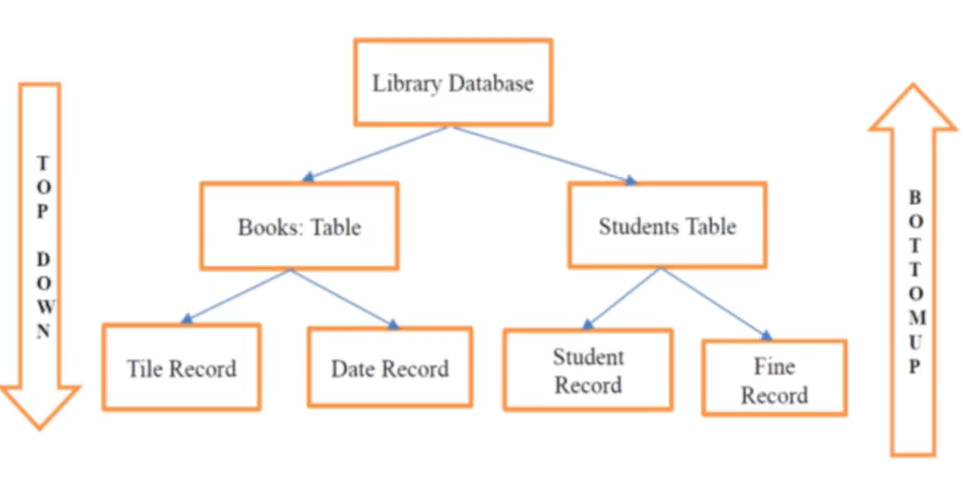
**Good software Design**

A good software design must be

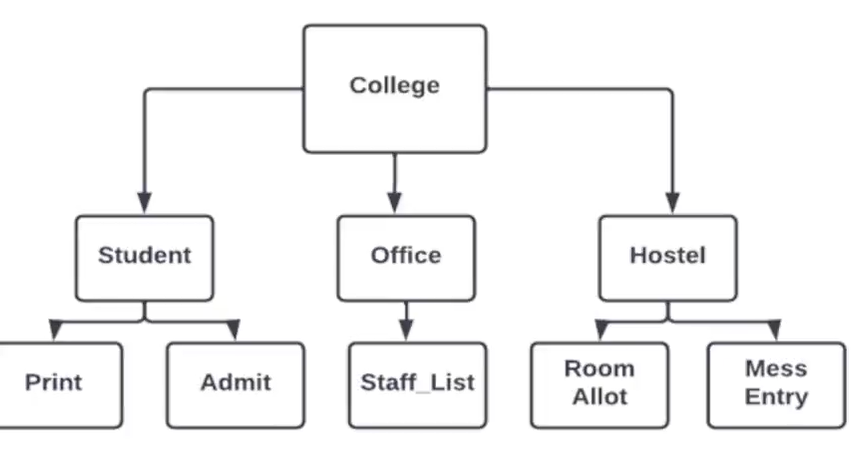
* Correct: Must meet all the requirement functional or non-functional.
* Modular: The system must be divided into modules.
* Reusable: Code for each module must be reusable in other modules or projects
* Maintainable: Must be easy to update or fix bugs.
* Simplicity: Desing must not be overly complex to understand.
* Scalability: Can handle growth of data, users etc.

**Software Design Principles**

1. Modularity: The system must be divided into different small modules making it easier to develop and test them individually. After development they must be easily integrated with each other to form complete system.
2. Abstraction: Focus on what the modules does rather than how it does it. This is hiding the complex details and only showing necessary interface.
3. Reusability: Ensure that each module is written in such a way that they can be reused in other program or same program with very little changes.
4. Portability: The system must be able to run on different platforms and operating systems.
5. Flexibility: The system must be flexible enough to accommodate changes with minimum changes to the modules.
6. Testability: The software should make it easy to test individual modules or the complete system

**Software Design Strategies**

**Top-down approach:** We start from the main system and break down the system in simpler sub-modules, good clarity and control.



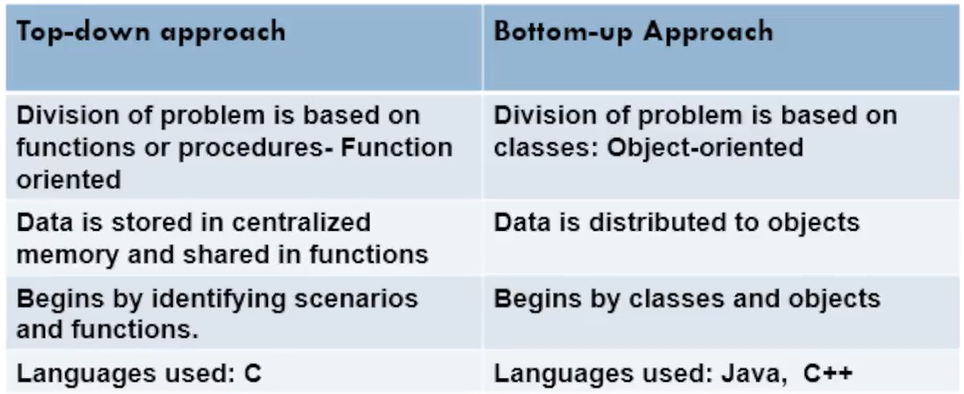
**Bottom-up approach:** We start from small modules and integrate them to form the complete system, useful when reusable component exist.

**Function oriented-design (Top-Down approach)**

* Start with overall system goal, then break it down into functions/procedures.
* Focus on what the system will do, not what data it handles.
* Useful when requirement are well defined

**Object oriented design (Bottom-Up approach)**

* Start by designing objects/classes (small, modules, reusable).
* Then combine them to build the complete system
* Focus is on data(objects) and their behaviour.
* Ideal for system that evolve overtime or use pre-existing components



**1. Top-Down Design**

* Start from the **main system** and break it into sub-modules.
* Refinement continues until each part is simple enough to be implemented directly.
* Example: Decomposing a banking system into account management, transactions, and reports.

✅ *Advantage:* Easy to understand  
❌ *Disadvantage:* Requires full understanding of the system at the start.

**2. Bottom-Up Design**

* Starts from the **basic components** (functions, classes) and builds up the system.
* Focuses on **reuse** of well-tested modules.

✅ *Advantage:* Encourages modular development  
❌ *Disadvantage:* System-wide integration can be challenging.

**3. Structured Design**

* Uses **Data Flow Diagrams (DFDs)** and **Structure Charts**.
* Emphasizes **logical structure**, **control flow**, and **modularity**.

✅ *Best for:* Transaction-based systems

**4. Object-Oriented Design (OOD)**

* System is designed using **classes**, **objects**, and **relationships** (inheritance, polymorphism).
* Models real-world entities and behavior.

✅ *Advantage:* Reusability, modularity, better handling of complexity  
❌ *Disadvantage:* Can be overkill for small projects

**5. Data-Oriented Design**

* Focuses on **data structures** and how data flows between functions/modules.
* Often uses **Entity-Relationship Diagrams (ERD)** or **Database schema-first** design.

✅ *Advantage:* Excellent for data-heavy systems  
❌ *Disadvantage:* May ignore user behavior or event-driven aspects

**Structure Charts**

A graphical representation of hierarchal structure of a software. It shows how system is broken down into modules and how modules are related.

**Notation**

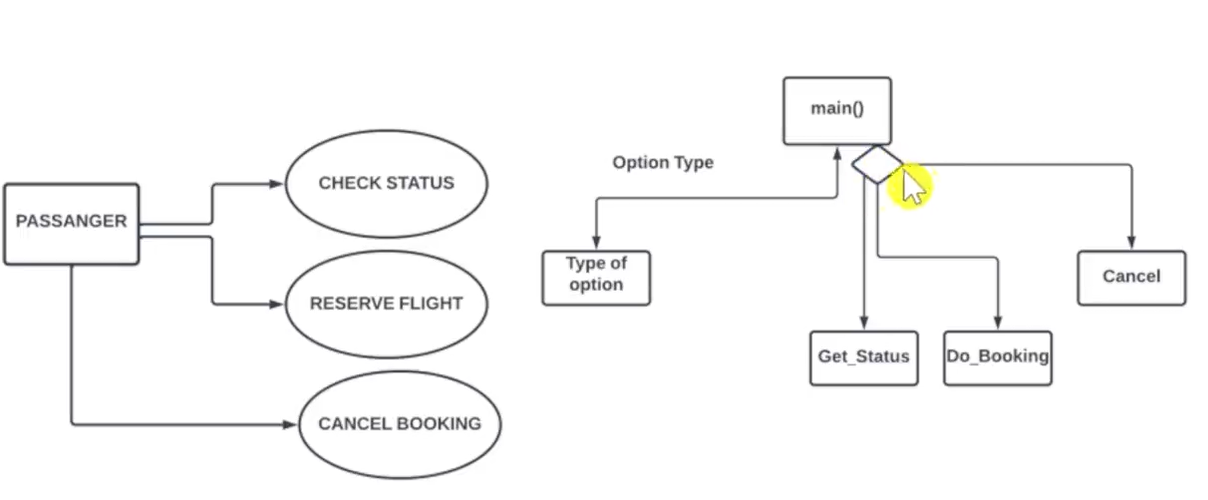
Modules are represented as rectangle

Library modules have rectangle with double edge

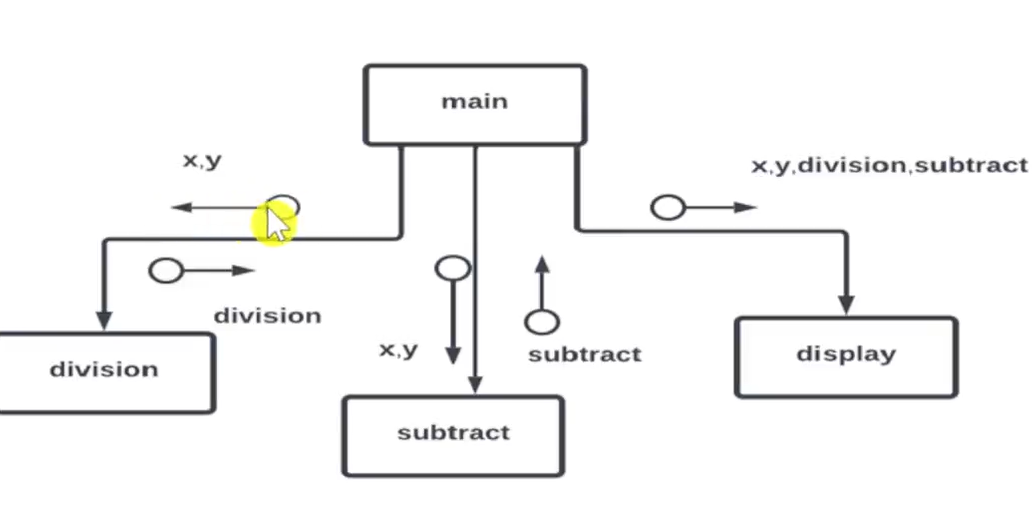
Arrow with notation may show data being passed between modules

Diamond for selection command and loops are represented using repetition around the diamond (control flow).

**DFD VS STRUCTURE CHART**



Structure chart for mathematical calculation



**Cohesion:** Refers to the fact that how closely the task in a single module related to each other. (high is best as it improves readability and maintainability)

High cohesion: tasks are closely related and module does one specific task.

Low cohesion: tasks are not that close and module does a lot of unrelated tasks.

**Types of cohesion:**

1. **Coincidental (lowest and Worst):** All random task are gathered in one module.Eg a module sends email, stores data and displays error message.
2. **Logical:** Task that perform similar task grouped together. Eg module handles all input output (read, write, print etc.).
3. **Temporal:** Tasks that are executed at the same time grouped. Eg booting of a system.
4. **Procedural:** Task following a sequence grouped together. Eg module takes input and processes it.
5. **Communicational:** Task that uses the same data grouped. Eg reading input and processing it.
6. **Sequential:** Tasks whose output is another’s input. Eg a module that extracts data → filters → writes to file
7. **Functional:** All task contributes to perform one single well-defined function. Eg calculateSlaray() does everything related to calculation of salary

**Coupling**: It defines that how much a module is dependent on other one for its own task. (low is good)

High coupling: module is heavily dependent on other.

Low coupling: module is loosely dependent on other and is almost independent.

**Types of Coupling:**

1. **Content(highest):** In this type a module changes code of another module directly. Eg function A jumping in function B’s code or changing its local variables. Worst type it affects modularity.
2. **Common:** Modules uses global variables changing them affects all the modules using these.
3. **Control:** Controlling another modules logic by passing some control variable such as mode=”even” to get random even numbers.
4. **Stamp:** Two or more modules using a composite DS .like structure in C
5. **Data(lowest):** Modules communicating with each other using minimum data possible. Eg giving only customer id to get the bank balance of the customer.

**How to achieve Functional Independence**

* A module with high cohesion and low coupling
* Must perform a well-defined single task without much interaction with other modules.

**Advantages**

* **Error Isolation:** If any error occurs it won’t directly affect other modules.
* **Scope of reuse:** A cohesive module can be easily taken out and reused.
* **Readability & Simplicity:** Makes the code readable and simple.
* **Improves modularity**

**Object Oriented Design (OOD)**

It is a conceptual model where system is view as a collection of objects. Object is an entity in a solution. Object is defined by a class and a class is abstract data type having attributes and methods. Object is instance of a class.

**Principles/Features**

1. **Abstraction**: Shows only essential features, hides complexity. Simplifies code usage and reduces unnecessary understanding burden.
2. **Encapsulation:** Bundles data and methods of same type into one unit (class) and hides internal details. Ensures data protection and prevents unintended access or modification.
3. **Inheritance**: Allows one class to acquire properties of another (parent-child). Promotes reusability and avoids code duplication.
4. **Polymorphism:** Same interface or method name behaves differently in different classes. Increases flexibility and code extensibility.
5. **Modularity:** Breaks the program into smaller, independent units (classes/modules). Makes the system easier to develop, test, and maintain.

# Development Phase: Language Selection, Coding Guidelines, Writing Code, Code Documentation

**🔹 1. Selecting a Programming Language**

**📌 Factors to Consider:**

* **Application Domain:** Web, mobile, embedded, scientific.
* **Team Expertise:** Skills and familiarity with the language.
* **Performance Requirements:** Speed and memory usage.
* **Platform Compatibility:** OS, hardware, integration.
* **Tool Support:** Availability of IDEs, libraries, frameworks.
* **Maintainability and Community Support:** Ease of hiring and long-term sustainability.

✅ *Example:*

* For system-level software → C/C++
* For web development → JavaScript, Python
* For enterprise apps → Java, C#

**🔹 2. Coding Guidelines**

**🧾 Key Practices:**

* **Consistent Naming Conventions:** Meaningful variable and function names.
* **Indentation and Formatting:** For better readability.
* **Avoid Hardcoding:** Use constants and configuration files.
* **Error Handling:** Include proper exception and validation mechanisms.
* **Code Comments:** For clarity of complex logic.
* **No Redundancy:** Follow DRY (Don't Repeat Yourself) principle.
* **Secure Coding:** Input validation, authentication, encryption.

✅ *Benefits:*  
Improves **code quality**, **readability**, and **collaboration** within teams.

**🔹 3. Writing the Code**

**🔧 Best Practices:**

* Follow the approved **design document**.
* Implement **unit tests** alongside development.
* Apply **modular programming**: break code into independent, testable functions/modules.
* Use **version control tools** (like Git).
* Regularly review and refactor code.

✅ *Goal:* Code should be **efficient**, **scalable**, and **easy to debug**.

**🔹 4. Code Documentation**

**📚 Types of Documentation:**

* **Inline Comments:** Clarify logic and decisions in the code.
* **API Documentation:** Explains function/class inputs, outputs, and behavior.
* **User Manuals:** For non-technical end users.
* **Technical Documentation:** Describes system architecture, data flow, dependencies, and setup.

✅ *Tools:* Javadoc, Sphinx (Python), Doxygen (C/C++)

✅ *Benefits:*  
Enables easier **onboarding**, **maintenance**, and **future upgrades**.

**Unit-4**

**Software Testing**

Software testing is the process of evaluating a software to check whether the software meets all the requirements without any errors or defects. It checks for completeness, correctness, performance and security.

During testing software is executed on sample data known as testcase.

**Objectives**

* Ensures that software runs properly without failure and meets all the user’s requirements.
* Improves quality and reliability of software.
* It detects any bugs, issues with the software.
* If there are any defects it can lead to security violations, system crashes, wrong computation, data loss or data leak.

**Terms**

**Bug:** When a software doesn’t work properly it is said to have a bug in informal language. Bug can be dude to algorithm, data storage, or resource etc.

**Defect:** When the system doesn’t meet the user requirement it is said to have a defect. Defect can be minor, major or critical.

**Error:** These are the errors in the code. They can be syntax, semantic, hardware error, testing errors etc.

**Fault:** When the system doesn’t work correctly its known as a fault. Fault can be in the interface, front-end, security or performance of the system etc.

**Failure:** When a system has numerous amounts of defects in it and these bugs affect the functionality its known as system failure condition.

**Verification (are we building the right project?)**

It ensures that the software being built meets the requirements of the user. It is a static process done without running the code. It is done through code review, code analysis, code walkthroughs and inspection. It is performed by designers and analyst. High-level deign and data base design is tested in the process.

**Validation (are we building the project right?)**

It is a dynamic process which involves running of the code. It makes sure that the user requirements are actually met with the user. Validation is performed after verification.

**Objective of V&V**

* Build up confidence in both users and developers.
* Ensure completeness and correctness of the system
* Find bugs and issues in the early stages of development
* Make sure that the project is fit for user requirements.

**Software inspection**

Software inspection is the process of official code review to find any bugs or issues in the software in early stages of development. The issues can be in the UML design, system architecture, code etc.

Software inspection is a part of verification hence can be done on incomplete code. It can not only check functionality of software but also the maintainability, portability.

**Objectives**

* To find out bugs in the early stages of dev so that they won’t go up on later stages where it would be costlier to fix them.
* Ensure coding standards are being followed.
* Improve software quality.
* Improve security of the software.

**Software testing process:**

Software testing process involves different series of steps such as planning, test case design, execution and reporting. It ensures system is functionally correct, meets requirements and is free of any major defects

**Phases:**

1. **Planning:** Define the objective, scope and approach of the test.
2. **Test case design:** Think of different scenarios, make a detailed document of testcases having inputs, steps to be followed and expected outputs.
3. **Set up environment:** Set up the hardware/software for testing. Make sure that the system is ready to be tested
4. **Execute test:** Execute each testcase manually or using an automated tool.
5. **Bug tracking and recording:** Track different bugs and issues. Assign the bugs to developers and retest after they are fixed.
6. **Test closure:** Document overall quality of the software, system failure and bug density

**Unit Testing**

It is the process of testing small specific components such as components, functions, classes in isolation to ensure that they work accordingly. It is done in development phase by the developers. The simplest way is to execute each and every line of code to check for loops, functions, etc. it focuses on logical correctness, input output validation of program.

**Advantages**

* Catches bugs in early phases of development
* Allows code changes in code
* Makes debugging easier
* Improves code quality and maintainability

**Integration Testing**

It is the process of testing the modules/components to ensure that they work together as expected. It is done after unit testing of each component is done. It ensures data flow from each module to another works without any error and detects if there are any errors related to interface or integration.

**Advantages**

* Detects interface defects.
* Ensures end-to-end data flow.
* Verifies that modules work as expected together.

**Types**

1. **Big Bang integration:** All the modules are combined at once and tested. Very simple to implement but very hard to debug. As in case of errors its hard to track which module is causing the error.
2. **Top-down integration:** Testing starts from the higher level of modules and integrates to lower levels. If lower modules are not ready program stubs are used instead. It is useful to test high-level interface and flow.
3. **Bottom-up integration:** Testing starts from the lower level of modules and moves toward upper levels. If upper modules are not ready drivers are used instead. It is suitable for projects where lower modules are ready and need thorough testing.
4. **Mixed integration:** It combines top-down and bottom up approach for more dynamic testing & thorough testing. Both stubs and drivers are used.

**Release Testing**

It is the final testing done before delivering the software to the customer. It checks the complete and integrated software works as expected or not. It ensures that all the user requirement are met by the software. Only functionality of software is tested rather than implementation.

**Advantages:**

* Ensures that all the requirement are met
* No bugs and integration errors are in the software.
* Tests software on real world scenario to make sure its ready to deploy.

**Types**

1. **Requirement-Based Testing:** The testcases are directly created from the SRS to make sure all the requirement stated by the user are strictly followed.
2. **Scenario-Based Testing:** Develops the testcases on basis of real world workflow, interaction. The test cases also include how a user will interact with the software.

**Continued……**

1. **Performance-Based Testing:** Tests the software’s performance, scalability and stability under expected and stress loads. Ensures that software is scalable and reliable in real world cases.

Ensures that software performs well under peak loads.

Detects bottlenecks, breaking point, slow response etc.

**Types**

* **Load Testing:** Tests the software under expected and peak loads. Eg 100 of users simultaneously accessing the software.
* **Stress Testing:** Pushes the software to its limits until failure. This is done mainly to observe how system fails and its recovery timing.
* **Soak Testing:** Tests the software under peak load for prolonged time. It is to check for any memory leaks and performance degradation.
* **Spike Testing:** Tests how the software reacts to sharp sudden increase in traffic.

**It also includes compatibility testing, recovery testing.**

1. **Regression Testing:** This testing is done after all the update, bug fixes etc. are done. It re-runs all the previous tests where bugs were found to ensure that no bugs are remaining. It also tests if there are any new bugs that are introduced to the system.

Ensure stability after changes.

Prevent old bugs from reappearing.

Maintain quality as the system evolves.

**System testing (User testing)**

System Testing is the final level of testing where the entire integrated system is tested as a whole against the requirements.

It checks if the system meets functional and non-functional expectations from a user's perspective.

* Validate the complete software system
* Ensure everything works together correctly
* Simulate real-world scenarios before release

**Types Involved:**

* **Functional Testing** – verifying features
* **Non-functional Testing** – performance, security, usability
* **End-to-End Testing** – full user workflows

**Alpha Testing**

* Performed **in-house** by internal employees or testers
* Happens **after system testing** but **before release**
* Simulates real user behavior in a **controlled environment**
* Focuses on **identifying major bugs and usability issues** early
* Helps improve software quality before it reaches actual users

**Beta Testing**

* Performed by **real users/customers** outside the organization
* Happens **after alpha testing** and just **before final release**
* Conducted in the **real-world environment**
* Collects **feedback on performance, usability, and unexpected issues**
* Helps catch bugs that might have been missed earlier

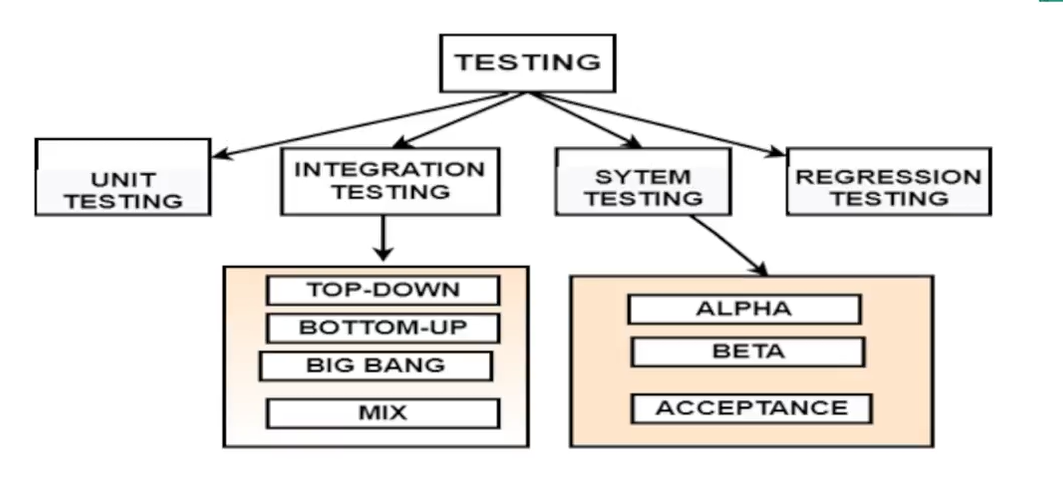
**Acceptance testing**

It is the final stage of testing, it is done by the customer to ensure that all the requirements are adopted and implemented in the project. It involves both functional and non-functional requirements.

Phases:

1. Acceptance criteria: An acceptance criteria formal document is singed between the customer and developers, which clearly states all the requirement of the software. These may vary from initial requirements.
2. Acceptance plan: A plan is made for acceptance with a budget, resources and testing.
3. Test cases: according to the plan the testcases are derived and it contains both functional and non-functional requirements.
4. Run Test: Now the test is executed as the plan states in client environment, some training is given to end users
5. Negotiation: Now the developers and users negotiate on test results whether to accept the software.
6. Final Decision: In this stage the developer and customer decides whether the software must be accepted or rejected. If the system is not as expected then further development is necessary to fix defects. Once its done the system is accepted.

**LEVEL OF TESTING**



**White Box testing**

White box testing aka clear-box testing/ code-based testing/glass box testing. In this type of testing the tester knows and understands the system code. The testcases are made on the basis of code and code coverage. It is made to test loops, functions and different component of system. It is used to check untested part of code and functionality of code.

White box testing includes

1. Statement coverage testing: This includes testcases so that each and every statement of the code can be executed at least once.
2. Branch coverage testing: This ensures checking of each branch of the conditional statements. The values of both true and false are tested.
3. Conditional testing: This test is for nested conditional statements. It ensures that every nested condition is working without error.
4. Data-flow based testing: This test identifies any errors such as undeclared variable errors, memory of variables deleted even though its being used in future etc.
5. Mutation testing: It is a technique to test the robustness of testcases as some line of source code is mutated (changed very little) to make sure testcases fails.

**Path testing**

Path testing uses testcases so that all the paths in a program are tested. Paths may be linearly independent.

A linearly independent path can be defined by using CFG (control Flow Graph).

* To determine the maximum number of linearly independent paths present in our software/code we can use McCabe’s cyclomatic complexity.
* Cyclomatic complexity determines the stability and confidence level of our software.
* It determines complexity of our software.
* It also determines the number of paths to be tested in our system/software.
* The number of decision statements (N) can be used to find the cyclomatic complexity (C) of a program (P). **C(P) = N+1**

**Black Box testing**

Blackbox testing is testing of functionality of a software based on given requirements. This is done without any knowledge of the design or the code of system. It is AKA behavioral testing.

**Steps:**

1. **Examine the requirements:** Study all the requirements expected from the software.
2. **Develop Testcases:** Develop test cases on the basis of requirements make sure to add inputs and expected outputs.
3. **Test:** Run the test record any errors, unexpected outputs.
4. **Fix and retest:** After fixing any errors, retest the system to ensure the errors don’t reappear or no new error must appear Repeat the process if required.

**Black Box testing strategies**

**Equivalence class partitioning:** This technique is used to minimize the number of possible built testcases. It divides the set of input values in different classes and test cases are made based on that. The test cases are built in such a way that testing one class also tests for other automatically. Eg testing if age is in range of 0-100 one class other is if its between 0-18 or 19-100. **Testcase: 20 it checks all the classes.**

**Boundary value analysis:** Here a software is tested for input boundaries. The general practice is to test the value from middle value to input range. Eg max 10 login allowed per day, start testing from 5 then 8 and then 10 on 11th it should generate an error.

**Decision Tables:** Method tests the input, test conditions and the result expected saved in form of decision tables.

**🔹 Test Drivers and Test Stubs**

**Test Drivers**:

* Temporary modules used in bottom-up testing.
* Simulate calling module for the unit being tested.

**Test Stubs**:

* Temporary modules used in top-down testing.
* Simulate the behavior of lower-level modules not yet developed.

**Importance**:

* Help in partial integration testing.
* Support modular and incremental development.

**🔹 Static Testing Strategies**

**🔸 Formal Technical Reviews (Peer Reviews)**

* Team members test and verify each other's documents to identify and correct the defects
* Helps catch issues early without executing code.

**🔸 Walkthroughs**

* Informal code review by the author with peers.
* Reviewer asks questions and gives feedback.
* Focuses on clarity and logic.

**🔸 Code Inspection**

* Formal, checklist-driven code review.
* More thorough than walkthroughs.
* Catches deviations from standards, logic flaws.

**🔸 Compliance with Standards**

* Verifies code against coding/design standards.
* Ensures consistency, readability, and maintainability.

**🔹 Automated Testing**

**Definition**:  
Use of tools to automatically run tests, check results, and compare them to expected outcomes.

**Examples of Tools**:

* Selenium (web)
* JUnit/TestNG (Java)
* PyTest (Python)

**Benefits**:

* Faster execution of large test suites.
* Better for regression testing.
* Ensures repeatability and consistency.

**Limitations**:

* High initial setup time.
* Not all tests (e.g., usability) can be automated.

# Unit 5: Software Maintenance and Software Project Management:

**Software Maintenance**

Software maintenance is the process of making changes to the software after delivery to the customer. These are generally made to:

* Accommodate major changes
* Fix bugs
* Performance improvement
* Adding new features
* Security Updates

**Types**

1. **Correctness maintenance:** It is related to fixing errors or bugs. When the software is delivered and deployed, some users report bugs if they found any. These bugs are needed to be fixed.
2. **Adaptive maintenance:** When a user changes the platform in which they are running their software, then adaptive maintenance is required. Suppose customer introduced new android, apple os in the customer environment.
3. **Perfective maintenance**: These are some features that might evolve the application. This is required when the changes and features are advanced. This focuses on removing unwanted features and improving user’s experience.
4. **Preventive maintenance**: Future problems in the software are prevented using some modifications and updates. It helps in increasing the life of software. The software code is optimized and prepared to handle the changes in environment.

**Reverse Engineering**

Reverse engineering is the process of analyzing system to understand its architecture, design, structure and its functionality. Usually when the SC is available but doc is missing.

Purpose

* Recover lost documentation
* Understand legacy systems
* Enable maintenance of a legacy system.
* Help with software re-engineering.

Steps:

1. Code is improved for better readability. Legacy system doesn’t have proper naming conventions and format.
2. Functionality is not changed but complex statements are simplified
3. Analyze code, different module and its components.
4. Extract system design

**Software reengineering**

It is a process where an existing software is modifying and reorganized for efficient maintenance. This helps in software evolution and adds value to product. It reduces the risk and cost of developing a new software.

**Drawbacks**

* Some software cannot be reengineered
* Some steps such as data management or modification of software architecture needs to be done manually which is tedious task
* Software reengineering is quite challenging

**Software Maintenance Cost**

Software maintenance cost is decided by ACT (Annual Change Traffic) due to new or modified requests. ACT is ->

ACT = Kloc addded + Kloc deleted / Kloc total

KlocAdded= measure of kilo lines added

KlocSubstracted= measure of kilo lines deleted

Kloc total = both +

Maintenance Cost = ACT\*Development Cost

Factors affecting software maintenance cost are:

Technical: changes in modules, programing language, logic of modules etc.

Non-Technical: Scope, Stability in hardware etc.

**Software Configuration**

Software Configuration refers to the state of source code, system design, interface, SRS, etc. used by developers in SDLC.

As these states can keep changing we need a configuration software.

**NEED**

* Required to store different versions and updates of software.
* When shared property of an object is changed all the sub sequent modules must be updated to avoid inconsistencies.
* Integrity might get messed up if a lot of developers try to manipulate same code.
* Provides a freeze option to lock stable version during development and testing

**Software Configuration Management Process (SCM)**

**Software Configuration Management (SCM)** is the process of **systematically controlling changes** to software to maintain **integrity, traceability, and consistency** throughout the software development life cycle.

**Objectives of SCM:**

* Track and control **versions of code and documents**
* Ensure **collaboration without conflict**
* Maintain **consistency after changes**
* Enable **rollback to stable versions** if needed

**Activities in SCM:**

1. **Configuration Identification**  
   → Identify items (code, docs, design) to be controlled
2. **Version Control**  
   → Manage changes and maintain versions (e.g., Git)
3. **Change Control**  
   → Evaluate, approve, and implement changes
4. **Configuration Auditing**  
   → Verify that changes are properly implemented
5. **Status Reporting**  
   → Keep track of current state, changes, and progress

**CASE (Computer Aided Software Engineering)**

CASE is a set of automated tools which can help us in several task such as coding, system design, documentation, etc.. Its objective is to automate and simplify such task for us, to ensure improved quality and productivity in software.

Software configuration and project management are also supported by CASE.

**Types**

**Upper CASE:** Upper CASE is used for initial levels of development such as requirement gathering, information collecting, system & architecture design

**Lower CASE:** Lower CASE is useful for later phases in development for coding, implementation and documentetation.

**Integrated CASE:** Covers all SDLC phases

**Benefits:**

* Reduces development time
* Reduces errors
* Better and structured documentation
* cost-saving,
* consistency and completeness in designs
* cost of development can be saved about 30-40%.

**Drawbacks:**

* Training required for developers to use CASE tools.
* Staff may resist use of CASE tools.
* Level of quality may not be desirable.
* Size of project may not be suitable

**Some Case tools:**

* **Requirement and Design Case Tool**
  + **StarUML:** A powerful UML design tool used to design software architecture using diagrams like class, sequence, and activity diagram.
* **Project Management Tool:**
  + **JIRA:** helps in tracking bugs, features, task mainly used in Agile and Scrum projects for planning and progress tracking.
* **Testing tool:**
  + **Selenium:** Used for automated testing of web applications. Supports many programming languages like java, python etc.

**Cost Estimation**

**COCOMO (Constructive Cost Model)**

COCOMO is a software cost estimation model developed by Barry Bohem. It factors in the efforts, labour, time, and cost required to develop a software project based on the sizes and type of project. The size project is measured In KLOC.

**Type of project**

1. **Organic:** Small and simple project with clear requirements and an experienced team. These projects are around 2KLOC-50KLOC.
2. **Semidetached:** Intermediate complexity project with a team of mixed experienced and inexperienced members. (50KLOC – 300KLOC).
3. **Embedded:** Large complex and real time system. The team of developers needs to know embedded system-based software. (>300KLOC).

Types of COCOMO model:

**Basic:** The basic COCOMO model gives an estimate of project parameters

Effort = a\* (KLOC)^a. PerMonth

Tdev = b\*(KLOC)^b Months

**Intermediate:**  This COCOMO also adds cost of drivers eg reliability, complexity, team experience.

**Detailed COCOMO:** adds phase wise effort distribution(like design, development, tetsing).

**Why use COCOMO?**

* Estimates **time, effort, cost** early in development
* Helps with **project planning and scheduling**
* Useful in **budget estimation** and **resource allocation**

**Risk Management in Software Engineering**

**Definition:**  
Risk management is the process of identifying, assessing, and controlling risks that may affect a software project's cost, schedule, scope, or quality.

**Steps in Risk Management:**

1. **Risk Identification:**
   * Detect potential risks that could negatively impact the project.
   * Examples: Unclear requirements, lack of skilled developers, technology changes, unrealistic deadlines.
2. **Risk Analysis:**
   * **Qualitative Analysis:** Assess the probability and impact of risks using descriptive terms (high, medium, low).
   * **Quantitative Analysis:** Use numerical values to estimate risk impact and likelihood.
3. **Risk Prioritization:**
   * Rank risks based on severity and likelihood.
   * Focus on high-impact, high-probability risks.
4. **Risk Planning (Mitigation & Contingency Planning):**
   * **Mitigation:** Steps taken to reduce the likelihood or impact of a risk.
   * **Contingency:** Backup plans if the risk occurs (Plan B).
5. **Risk Monitoring and Control:**
   * Continuously track identified risks.
   * Identify new risks during project execution and take corrective actions.

**Types of Risks in Software Projects:**

* **Technical Risks:** Issues with technology, software architecture, or integration.
* **Project Management Risks:** Poor planning, scope creep, or resource mismanagement.
* **Organizational Risks:** Changes in company strategy, loss of key personnel.
* **External Risks:** Market changes, legal regulations, or third-party failures.

**SEI-CMM (Software Engineering Institute - Capability Maturity Model)**

**Definition:**  
SEI-CMM is a **framework developed by the Software Engineering Institute (SEI)** at Carnegie Mellon University to assess and improve the **maturity of software development processes** in an organization. It helps organizations **develop high-quality software in a predictable and cost-effective manner**.

**Purpose of SEI-CMM:**

* To improve **software process capability**.
* To help organizations move from **ad hoc, chaotic development** to a more **disciplined and mature process**.
* To evaluate and benchmark the **software process maturity** of organizations.

| **Level** | **Name** | **Description** |
| --- | --- | --- |
| **1** | Initial | - Process is **ad hoc** and chaotic.- Success depends on **individual effort**. |
| **2** | Repeatable | - Basic project management processes are established.- **Repeatable successes**. |
| **3** | Defined | - Processes are **documented, standardized, and integrated** organization-wide. |
| **4** | Managed | - Processes are **measured and controlled** using metrics. |
| **5** | Optimizing | - Focus on **continuous process improvement**.- Uses feedback and innovation. |

**Key Process Areas (KPAs):**

Each maturity level (except Level 1) has associated KPAs. For example:

* **Level 2 KPAs**: Requirements Management, Project Planning, Quality Assurance.
* **Level 3 KPAs**: Organizational Process Focus, Training Programs.
* **Level 4 KPAs**: Quantitative Process Management.
* **Level 5 KPAs**: Defect Prevention, Process Change Management.

**Benefits of SEI-CMM:**

* Improved software quality.
* Better project predictability (cost, schedule).
* Enhanced customer satisfaction.
* Better risk management