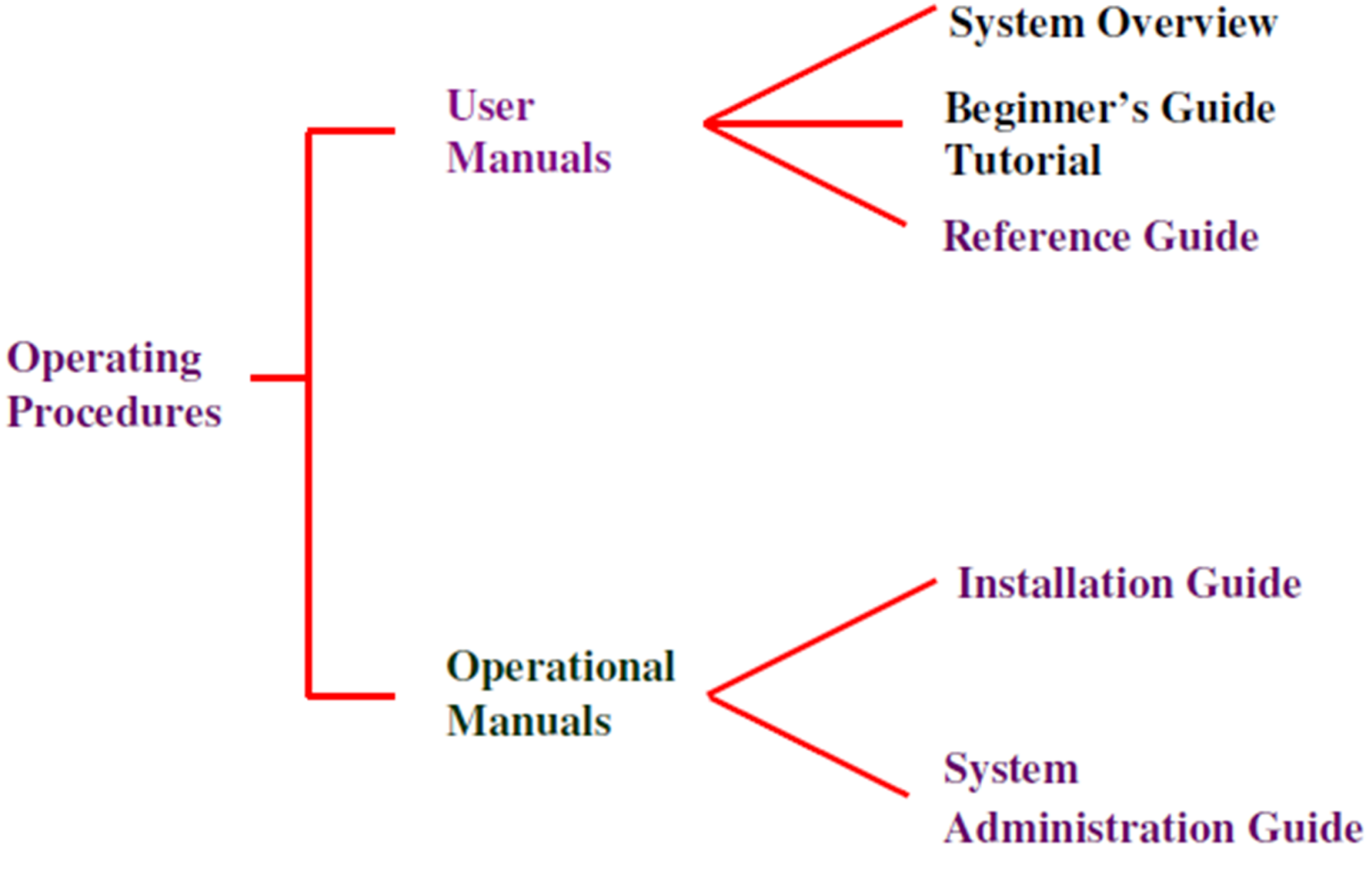
Module No.	Content	Teaching Hours
I	**Introduction:** Introduction to Software Engineering, Software characteristics, Software Crisis, Software Engineering Process. **Software Development Life Cycle (SDLC) Models:** Waterfall, Incremental, Iterative Enhancement, Prototype, RAD and Spiral Models. **Software Requirements Engineering:** Types of Requirements, Requirement Elicitation Techniques Like Interviews, FAST & QFD, Use case Approach, Requirements Analysis Using DFD, Data Dictionaries & ER Diagrams, Requirements Documentation, and SRS. **Software Project Planning:** Size Estimation like Lines of Code & Function Count, Cost. **Estimation Models:** COCOMO (Basic, Intermediate) **Software Design:** Cohesion & Coupling, Classification of Cohesion & Coupling, Function Oriented Design, Object Oriented Design, Structure chart. **Coding:** Characteristics of Coding and Coding style.	20
II	**Software Metrics:** Software Measurements, Token Count, Halstead Software, Measures. **Software Reliability & Quality:** Introduction of Mc Call's & Boehm's Quality Model, Capability Maturity Models **Software Reliability Models**: Basic Execution Time Model. **Software Testing:** **Testing Fundamentals:** Test Case Design, Black Box Testing Strategies, White Box Testing, Unit Testing, Integration Testing, System Testing. **Introduction to Automation Testing and Testing Tools:** Automated Testing Process, Framework for Automation Testing, Introduction to Automation Testing Tool. **Software Maintenance:** Maintenance Process **Maintenance models**: Belady and Lehman Model, Boehm Model Regression Testing, Software Configuration Management; Implementation, Introduction to Reengineering and Reverse Engineering. **Software Risk Management:** Risk Identification and Risk Analysis	20

INTRODUCTION TO SOFTWARE ENGINEERING

SOFTWARE

- Anything, that can be stored electronically.
- It is more than just a program code.
- It is considered to be **collection of executable programming code, associated libraries and documentations**.
- Software when made for a specific requirement is called "**Software Product**".
- Software refers to a set of instructions or programs that are written to perform specific tasks on a computer or a similar device. It encompasses the entire set of programs, procedures, and associated documentation related to the operation of a computer system. **Software is a critical component of modern computing and is essential for enabling the functionality of hardware.**



List of operating procedure manuals.

Difference between Software and Program

	Program	Software
1	Programs are developed by individuals for their personal use.	A software is usually developed by a group of engineers working in a team.
2	Usually small in size	Usually large in size
3	Single user	Large number of users
4	Lacks proper documentation	Good documentation support
5	Lack of user interface	Good user interface
6	Have limited functionality	Exhibit more functionality

SOFTWARE ENGINEERING

- Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

WHY STUDY SOFTWARE ENGINEERING?

- Computer information and control systems have integrated into the fabric of human society.
- They control our clocks, washing machines, motor vehicles, traffic lights, the electric power to our homes etc.
- It is the discipline dedicated to the principles and techniques required for the construction of the computer systems of today and tomorrow.

NEED OF SOFTWARE ENGINEERING

- The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

- ✓ Large Software
- ✓ Scalability
- ✓ Cost
- ✓ Dynamic Nature
- ✓ Quality Management
- ✓ Risk Management

CAREER OPPORTUNITIES

- Software engineers are among the highest paid professionals in most countries of the world.
- They are in demand in not only at software development companies but also in all other organizations that are involved in the development of significant information systems.
- As technology continues to advance, the demand for skilled software professionals remains high.

SOME TERMINOLOGIES

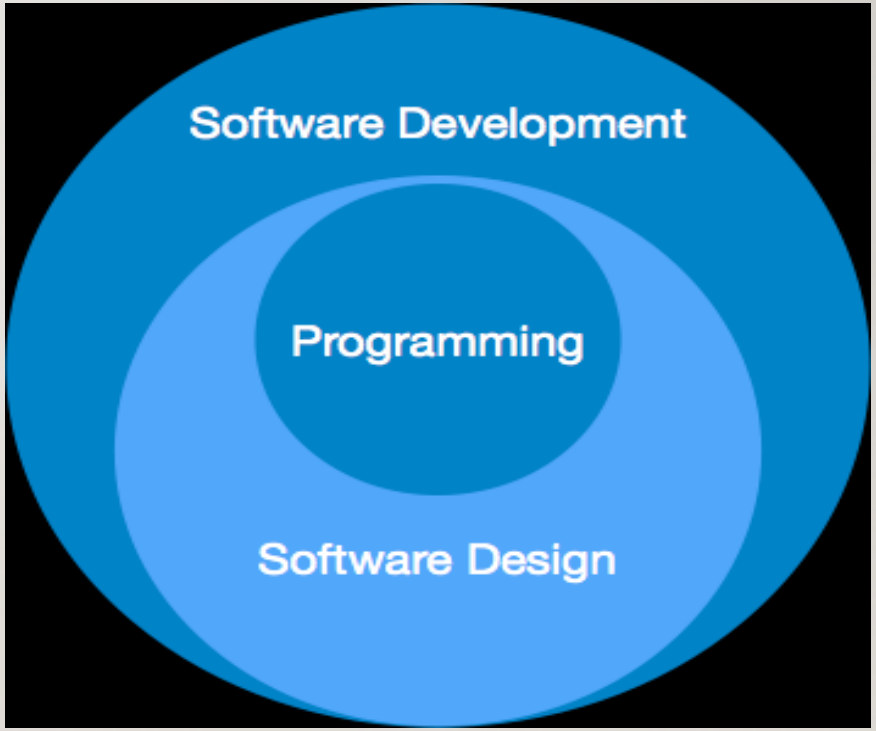
- **Product:** What is delivered to the customer, is called a product. It may include source code, specification document, manuals, documentation etc. Basically, it is nothing but a set of deliverables only.
- **Process:** Process is the way in which we produce software. It is the collection of activities that leads to (a part of) a product. An efficient process is required to produce good quality products. If the process is weak, the end product will undoubtedly suffer, but an obsessive over reliance on process is also dangerous.

SOFTWARE EVOLUTION

- The process of developing a software product using software engineering principles and methods is referred to as **software evolution.** This includes the initial development of software and its maintenance and updates, till desired software product is developed, which satisfies the expected requirements.

SOFTWARE PARADIGMS

- Software paradigms **refer to the methods and steps, which are taken while designing the software.** There are many methods proposed and are in work today, but we need to see where in the software engineering these paradigms stand. These can be combined into various categories, though each of them is contained in one another.



CONTD.

- **Software Development Paradigm**
  - Requirement gathering
  - Software design
  - Programming
- **Software Design Paradigm**
  - Design
  - Maintenance
  - Programming
- **Programming Paradigm**
  - Coding
  - Testing
  - Integration

CHALLENGES IN LARGE PROJECT

- Effort intensive
- High cost
- Long development time
- Changing needs for users
- High risk of failure – user acceptance, performance, maintainability

SUCCESSFUL SOFTWARE SYSTEM

- Software development projects have not always been successful,
- When we consider a s/w application successful?

❖Development completed

❖It is useful

❖It is usable

❖It is used

- **Usable, cost-effective and maintainable software product is called successful.**

SOFTWARE CHARACTERISTICS

Software characteristics refer to the inherent qualities and attributes that define the nature and behavior of software. Understanding these characteristics is essential for designing, developing, and maintaining effective and reliable software.

- Software becomes reliable over time instead of wearing out.
- It may be retired due to environmental changes, new requirements, new expectations etc.
- When a hardware component wears out, it is replaced by a spare part. But, there are no software spare parts.
- Hence, software maintenance require more complexity than hardware maintenance.
- Software is not manufactured, it is developed.
- Reusability of components.
- Software is flexible.

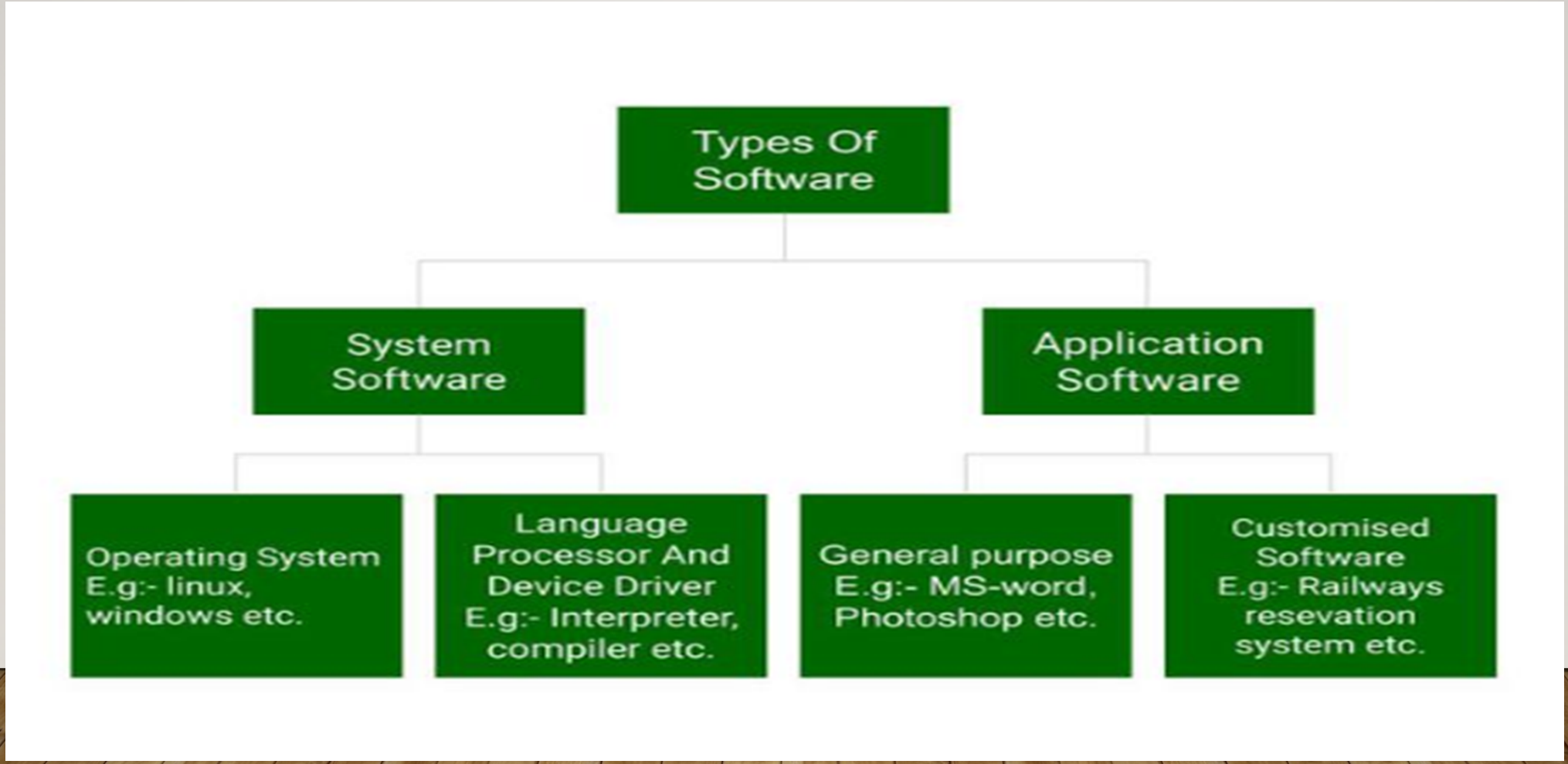
SOFTWARE CRISIS

- The term "software crisis" refers to a period in the early history of software development **when the challenges of creating complex software systems became apparent and were perceived as a crisis.**
- A software crisis is **a mismatch between what software can deliver and the capacities of computer systems, as well as expectations of their users.**

FACTORS CONTRIBUTING TO THE SOFTWARE CRISIS

- Schedule Slippage / late delivery
- Schedule Overruns and Cost Overruns
- Does not solve user’s problem
- Complexity of Software
- Poor quality
- Product does not meet specified requirements
- Lack of adequate training in software engineering
- Low productivity improvements.
- Lack of Standardization and Methodologies

TYPES OF SOFTWARE



SYSTEM SOFTWARE

- **Operating Systems (OS):** An operating system (OS) is a software program that acts as an intermediary between computer hardware and the computer user. It provides a set of services and manages system resources to ensure efficient and secure operation of the computer. The operating system serves as the foundation for other software applications and provides a user interface for interacting with the computer. Examples include **Windows, macOS, Linux, and Android**. They manage hardware resources and provide essential services for other software.
- **Device Drivers:** Device drivers are software components that facilitate communication between the operating system and hardware devices. They enable the operating system to interact with specific hardware components, ensuring proper functionality and allowing applications to use the devices. Examples **Graphics Card Driver, Printer Driver, Network Interface Card (NIC) Driver, Sound Card Driver.**

APPLICATION SOFTWARE

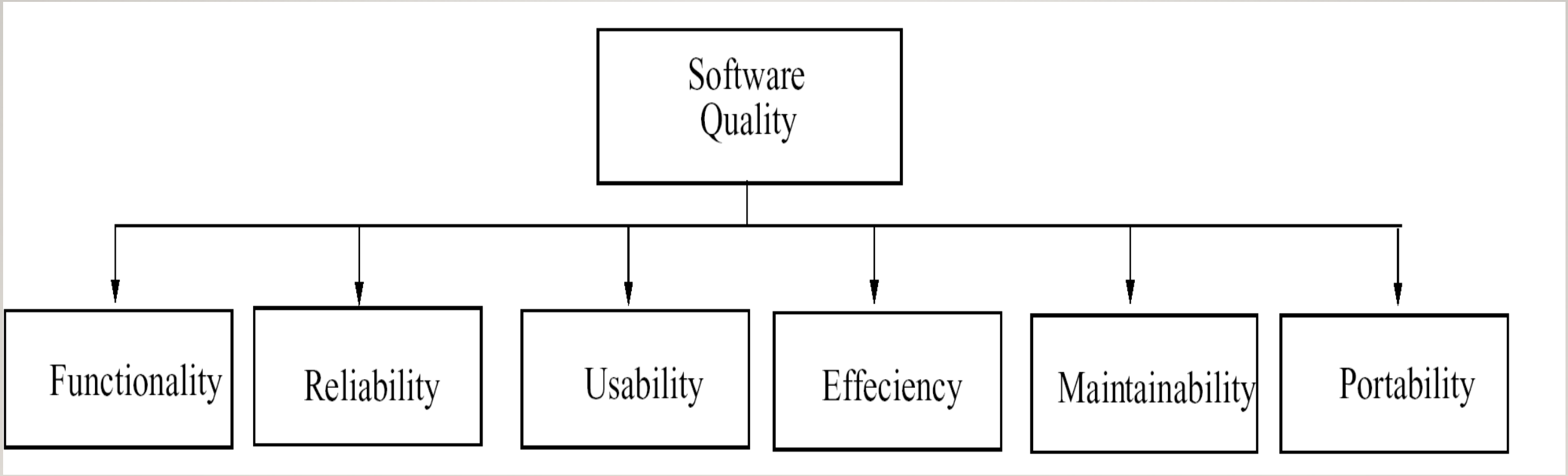
Application software, often referred to as **"apps" or simply "applications**," is a type of computer software designed to perform specific tasks or functions for end-users. Unlike system software, which provides the foundational services and manages hardware resources, application software is developed to meet the diverse needs and requirements of users.

- **Productivity Software:** Tools that help users perform tasks, such as **word processors (Microsoft Word), spreadsheets (Microsoft Excel), and presentation software (Microsoft PowerPoint).**
- **Web Browsers:** Software for accessing and navigating the internet, such as **Google Chrome, Mozilla Firefox, and Safari.**

CONTD.

- **Media Players:** Applications for playing audio and video files, like **VLC Media Player and Windows Media Player.**
- **Graphics Software:** Used for creating and editing images and graphics, such as **Adobe Photoshop and CorelDRAW.**
- **Database Management Systems (DBMS):** Software for managing and organizing data, including **MySQL, Microsoft SQL Server, and Oracle Database**.
- **Communication Software:** Facilitates communication, including **email clients (Microsoft Outlook), messaging apps (Slack, WhatsApp), and video conferencing tools (Zoom).**

SOFTWARE QUALITY ATTRIBUTES



CONTD.

- **Functionality** – The capability to provide functions which meet stated and implied needs when the software is used. It include suitability (appropriate set of functions provided), accuracy and security.
- **Reliability** – The capability to maintain a specified level of performance.
- **Usability** – The capability to be understood, learned and used. It includes understandability, learnability and operability.
- **Efficiency** – The capability to provide appropriate performance relative to the amount of resources used.
- **Maintainability** – The capability to be modified for purposes of making corrections, improvements or adaptation. It includes changeability, testability, stability etc.
- **Portability** – The capability to be adapted for different specified environments without applying actions or means other than those provided for this purpose in the product. It also include adaptability, install ability.

SOFTWARE PROCESSES:

- A software process encompasses a series of tasks and their corresponding results aimed at creating a software product, typically executed by software engineers.

There are four fundamental process activities inherent in all software processes:

- **Software Specifications:** It is essential to articulate the operational parameters and functionalities of the software.
- **Software Development**: It is necessary to generate software that aligns with specified requirements.
- **Software Validation:** Validation processes are crucial to verifying that the software aligns with customer expectations.
- **Software Evolution:** The software should undergo adaptation to address evolving client requirements.

SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

- A software life cycle model, also known as a process model, serves as a visual and diagrammatic depiction of the various stages a software product undergoes throughout its life cycle.
- This model encompasses the methods necessary for guiding the software product through its developmental phases and outlines the framework within which these methods are to be executed.

NEED OF SDLC:

- Without using an exact life cycle model, the development of a software product would not be in a systematic and disciplined manner. When a team is developing a software product, there must be a clear understanding among team representative about when and what to do. Otherwise, it would point to chaos and project failure.
- A software life cycle model describes entry and exit criteria for each phase. A phase can begin only if its stage-entry criteria have been fulfilled. So without a software life cycle model, the entry and exit criteria for a stage cannot be recognized. Without software life cycle models, it becomes tough for software project managers to monitor the progress of the project.

STAGE 1: PLANNING AND REQUIREMENT ANALYSIS

- The most crucial phase in the Software Development Life Cycle (SDLC) is Requirement Analysis. This step involves senior team members working closely with stakeholders and industry experts to gather essential information. It also includes planning for quality assurance and identifying potential risks associated with the project.
- During this stage, a meeting is arranged between the business analyst, project organizer, and the client. The goal is to collect all necessary data, such as what the customer wants to create, who the end users will be, and the overall objective of the product. Before diving into product development, it's vital to have a solid understanding and knowledge of what the product needs to achieve.

STAGE 2: DEFINING REQUIREMENTS

- Once the requirement is understood, the **SRS (Software Requirement Specification)** document is created. The developers should thoroughly follow this document and also should be reviewed by the customer for future reference.
- **"SRS"- Software Requirement Specification document which contains all the product requirements to be constructed and developed during the project life cycle.**

STAGE 3: DESIGNING THE SOFTWARE

- The next phase is about to bring down all the knowledge of requirements, analysis, and design of the software project. This phase is the product of the last two, like inputs from the customer and requirement gathering.

STAGE 4: DEVELOPING THE PROJECT

- In this phase of SDLC, the actual development begins, and the programming is built. The implementation of design begins concerning writing code.
- Developers have to follow the coding guidelines described by their management and programming tools like compilers, interpreters, debuggers, etc. are used to develop and implement the code.

STAGE 5: TESTING

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

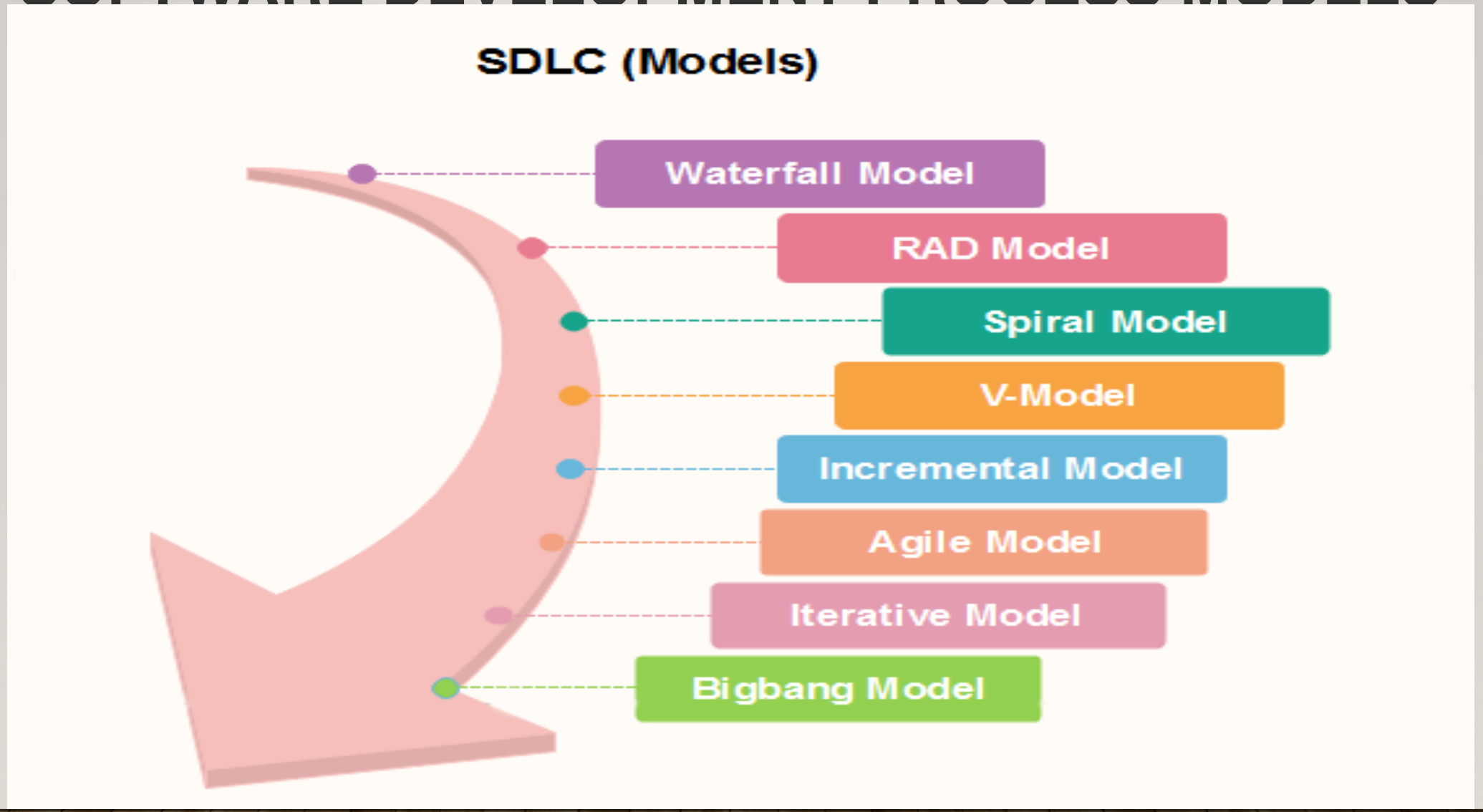
STAGE 6: DEPLOYMENT

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

STAGE 7: MAINTENANCE

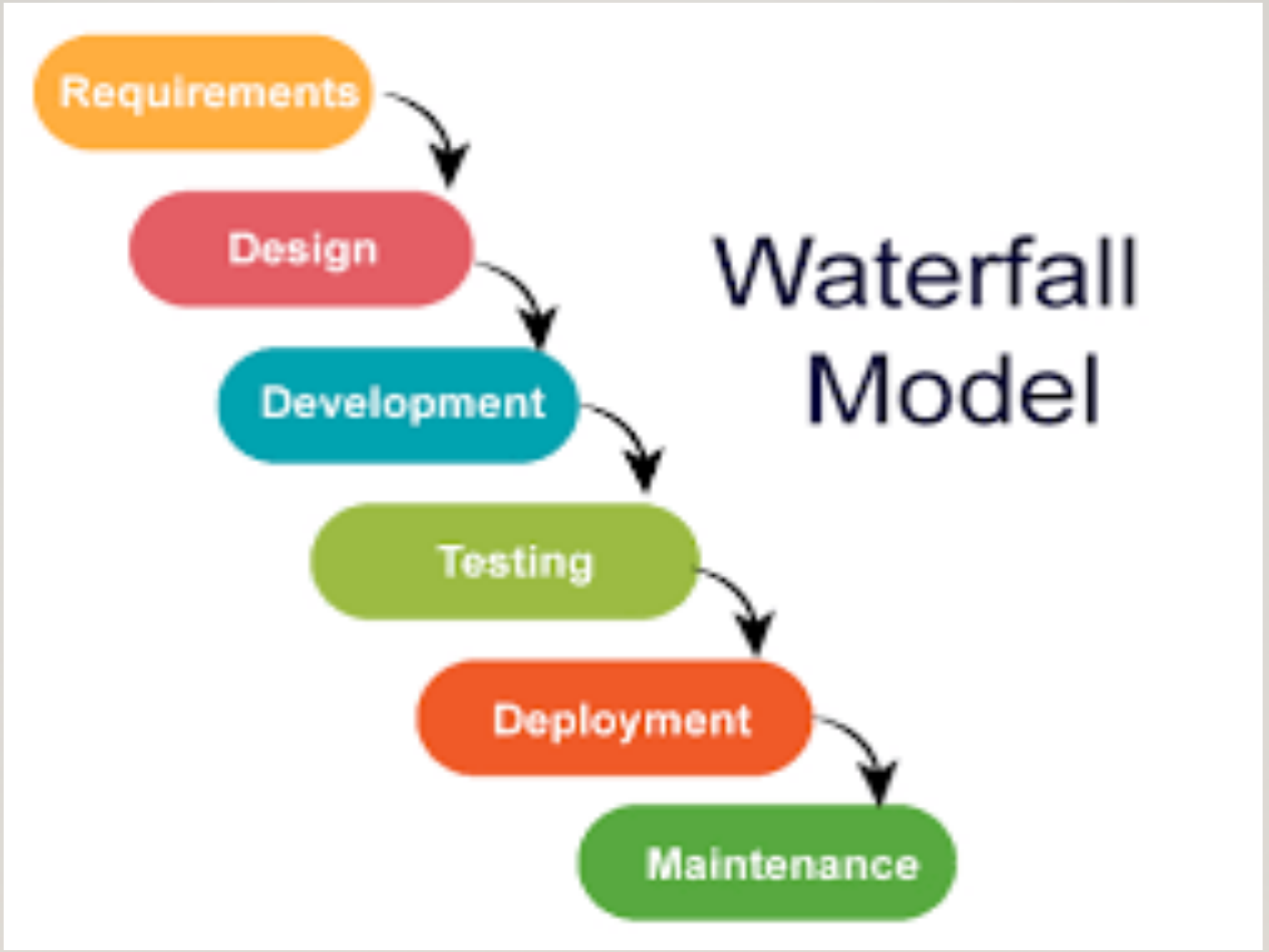
- Once when the client starts using the developed systems, then the real issues come up and requirements to be solved from time to time.
- This procedure where the care is taken for the developed product is known as maintenance.

SOFTWARE DEVELOPMENT PROCESS MODELS



WATERFALL MODEL

- The waterfall model is a linear, sequential approach to the software development lifecycle (SDLC) that is popular in software engineering and product development.
- The waterfall model follows a step-by-step approach in the Software Development Life Cycle (SDLC), resembling the way water flows over a cliff. Each phase in the development process has clear endpoints or goals, similar to distinct steps in a waterfall.
- **Importantly, once a phase is completed and its goals are achieved, they cannot be revisited or changed. The process flows in a linear fashion, with each phase building upon the outcomes of the previous one.**


Requirements
Design
Development
Testing
Deployment
Maintenance
Waterfall Model

Contd.

- The waterfall model doesn't include a project's end user or client as much as other development methodologies. Users are consulted during the initial stages of gathering and defining requirements, incorporating client feedback after that.
- By leaving the client out of the main part of the waterfall process, the development team moves quickly through the phases of a project.
- This methodology is good for teams and projects that want to develop a project according to fixed or unchanging requirements set forth at the beginning of the project.
- Waterfall projects have a high degree of process definition with little or no output variability. Waterfall is also a good choice if the project is constrained by cost or time.

ADVANTAGES OF THE WATERFALL MODEL

- Simple and easy to understand and use
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Clearly defined stages.
- Easy to arrange tasks.
- Process and results are well documented.

DISADVANTAGES OF THE WATERFALL MODEL

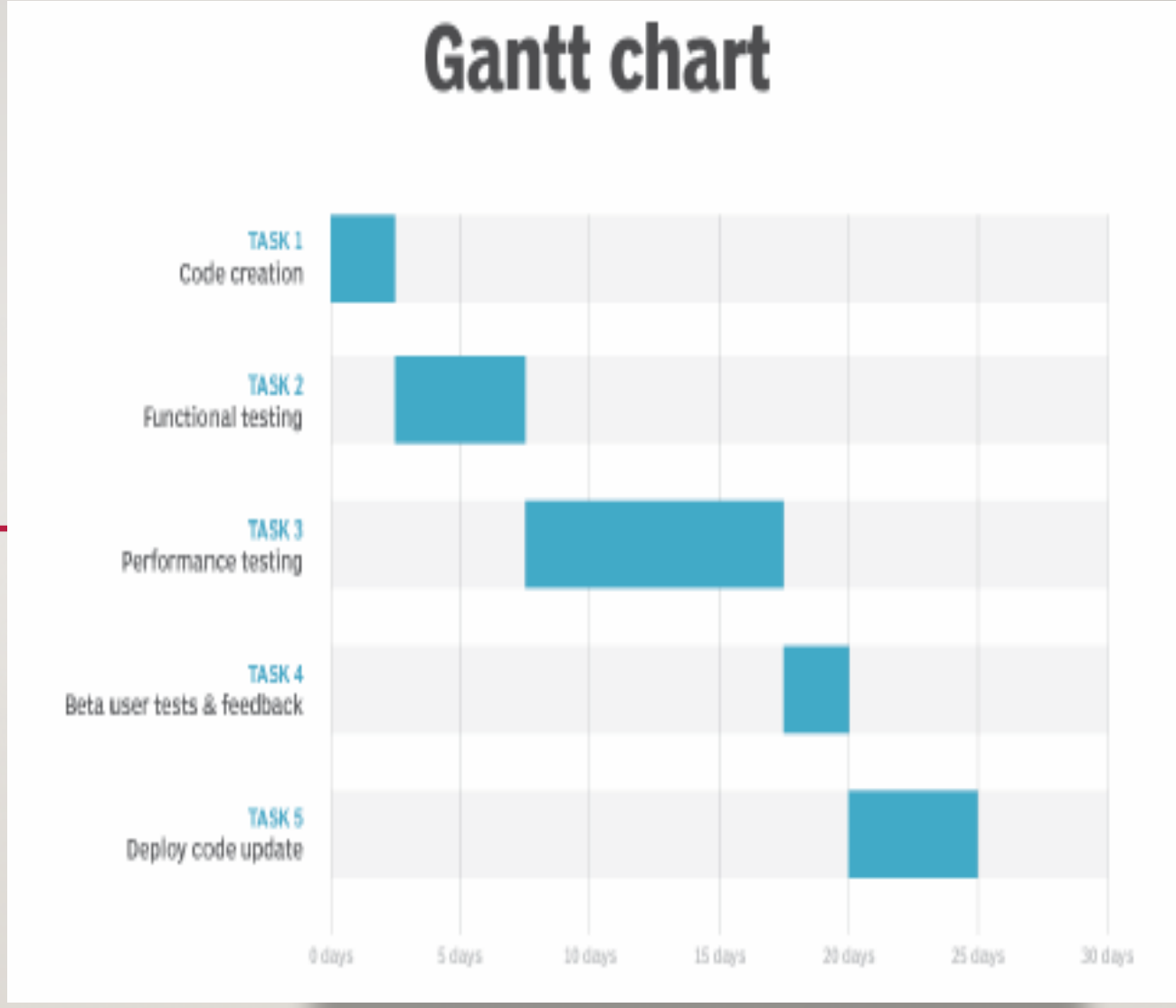
- Design isn't adaptive; when a flaw is found, the entire process often needs to start over.
- Method doesn't incorporate mid-process user or client feedback, and makes changes based on results.
- Waterfall model delays testing until the end of the development lifecycle.
- The methodology doesn't handle requests for changes, scope adjustments and updates well.

Contd.

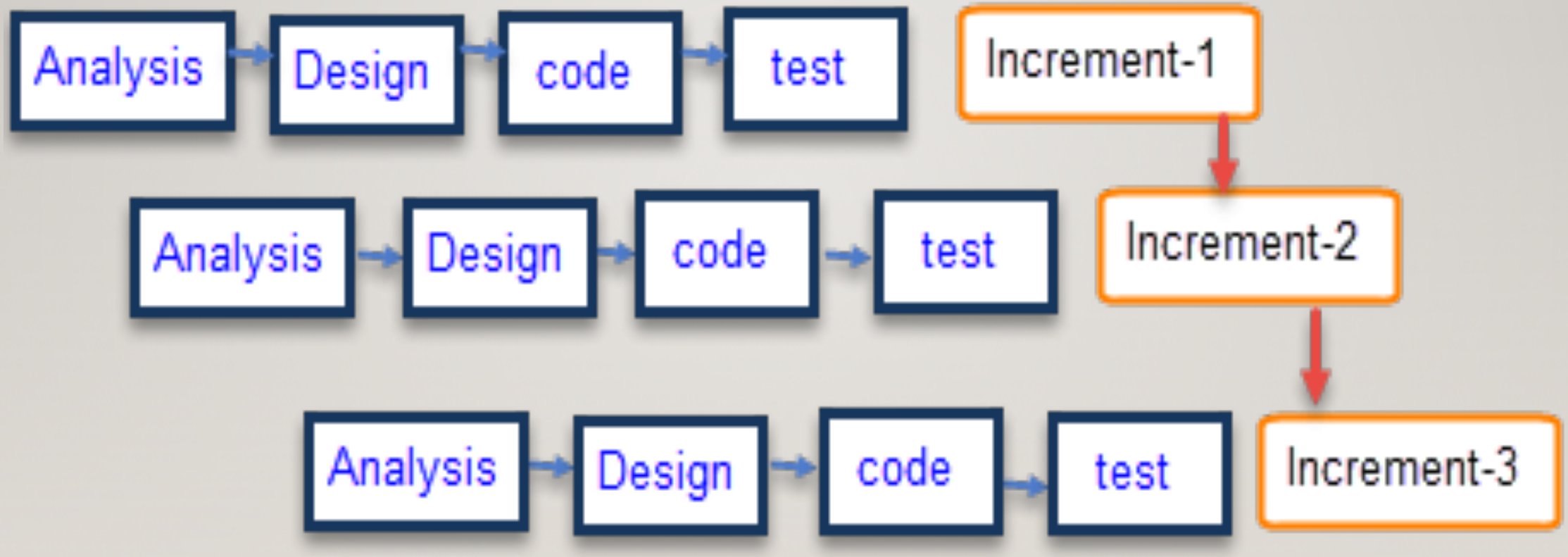
- Waterfall doesn't let processes overlap for simultaneous work on different phases, reducing overall efficiency.
- No working product is available until the later stages of the project lifecycle.
- Waterfall isn't ideal for complex, high-risk ongoing projects.

GANTT CHARTS

Gantt charts are a common management tool for waterfall projects. These charts enable easy visualization of sequential phases, letting project managers map dependencies and subtasks to each phase of the process. They provide a clear view of timelines and deadlines for each phase.



INCREMENTAL MODEL



Contd.

- Each iteration passes through the **requirements, design, coding and testing phases**. And each subsequent release of the system adds function to the previous release until all designed functionality has been implemented.
- The system is put into production when the first increment is delivered. The first increment is often a core product where the basic requirements are addressed, and supplementary features are added in the next increments.
- Once the core product is analyzed by the client, there is plan development for the next increment.

ADVANTAGES OF INCREMENTAL MODEL:

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

DISADVANTAGES OF INCREMENTAL MODEL:

- Needs good planning and design.
- Needs a clear and complete definition of the whole system before it can be broken down and built incrementally.
- Total cost is higher than waterfall.
- Well defined module interfaces are needed.
- Correction of a fault in one unit necessitates revision in other units and takes a long time.

WHEN TO USE THE INCREMENTAL MODEL:

- This model can be used when the requirements of the complete system are clearly defined and understood.
- Major requirements must be defined; however, some details can evolve with time.
- There is a need to bring a product to the market early.
- A new technology is being used
- Resources with needed skill set are not available
- There are some high risk features and goals.

POINTS TO REMEMBER:

- In the incremental model, each unit (known as an increment) is **intended to be a working model with complete, usable functionality**. These increments typically build upon each other, **gradually adding new features or functionalities until the final product is achieved.**

➢Initial planning

➢Developing increments

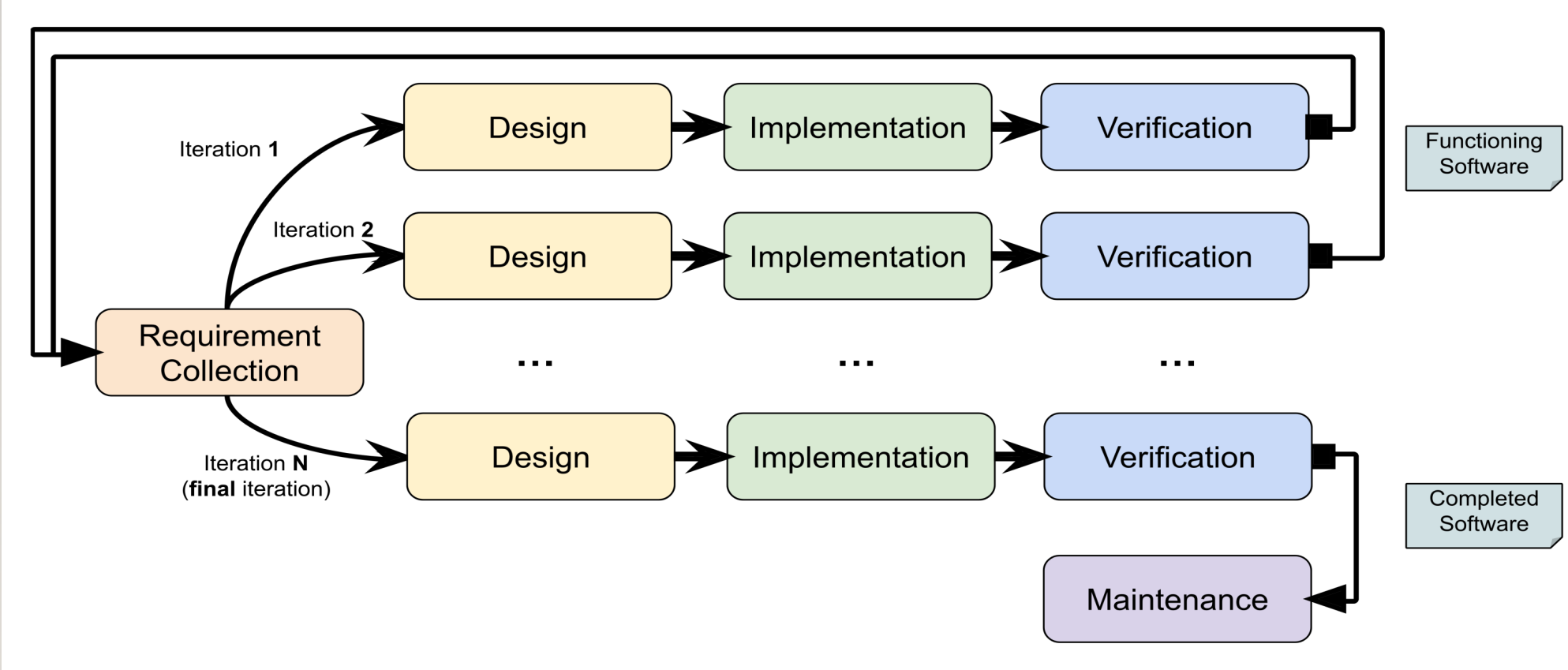
➢Delivery and feedback

➢Iterative refinement

KEY CHARACTERISTICS OF THE INCREMENTAL MODEL:

- Early delivery of working features
- Reduced risk
- Flexibility
- Potentially higher initial cost

ITERATIVE ENHANCEMENT MODEL



Contd.

- The terms **"iterative enhancement"** and **"incremental model"** are often used interchangeably, but there are some subtle differences between them.
- **Both deal with approaching a project in smaller steps, but the focus and delivery of those steps differ slightly.**
- Here, **focus** is on **Continuous improvement on a core functionality**. Each iteration builds upon the previous one, refining and enhancing existing features based on feedback and learning.
- **Delivery**: Working product delivered early on, with **each iteration adding new features or improving existing ones**. The final product is reached through successive enhancements.
- **Example: Building a mobile app. The first iteration might just be basic login and navigation. Subsequent iterations add features like user profiles, search functionality, and social media integration.**

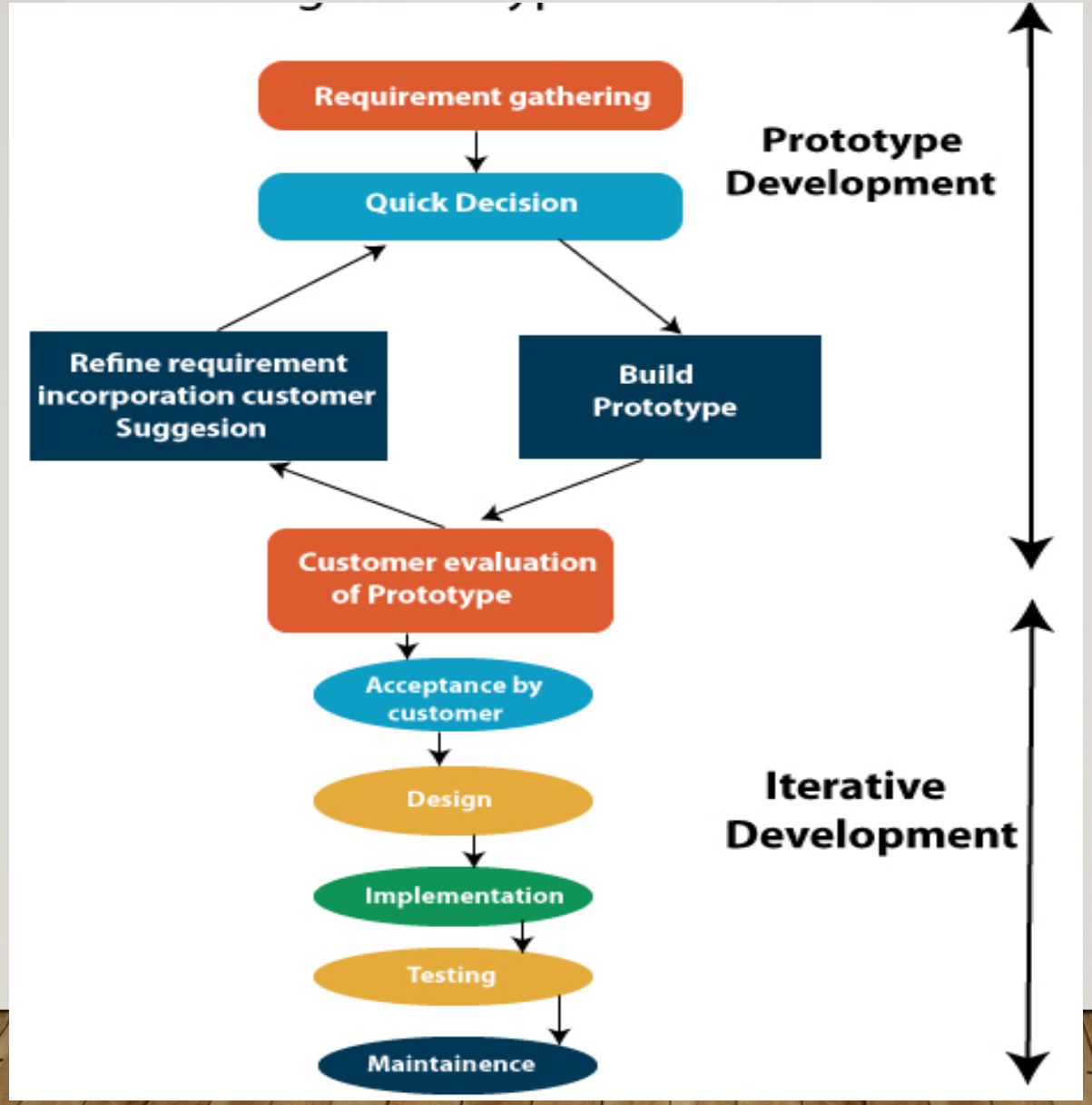
WHEREAS IN INCREMENTAL MODEL…….

- **Focus**: **Delivery of complete, smaller pieces of functionality in sequence**. Each increment adds a new, working feature to the overall product.
- **Delivery**: Features are delivered in stages, **with each stage building upon the previous one to eventually form the complete product. Early stages may not be fully functional**.
- **Example:** Implementing a **financial management system. The first increment might be basic account management. Subsequent increments add features like transaction tracking, reporting, and investment management.**

SUMMARIZING THE KEY DIFFERENCES:

Feature	Iterative enhancement	Incremental model
Focus	Continuous improvement on existing features	Delivery of complete, smaller features
Delivery	Early working product with improvements in each iteration	Features delivered in stages, building towards final product
Flexibility	Highly adaptable to changes and feedback	Less flexible, changes usually in subsequent increments
Risk	Less initial risk as features are added gradually	Higher initial risk as early stages may not be fully functional

PROTOTYPE MODEL



Contd.

- Emphasizes **building and refining a working model of the software product early** in the development cycle.
- It's an **iterative process** where user feedback and testing guide the development of the final product.
- Build a basic, initial version of the software (prototype) and continuously improve it based on user feedback and testing.

➢ **Requirement Gathering and Analyst**

➢ **Quick Decision**

➢ **Build a Prototype**

➢ **Assessment or User Evaluation**

➢ **Prototype Refinement**

➢ **Engineer Product**

ADVANTAGES OF PROTOTYPE MODEL

- Reduced risk
- Early user feedback
- Increased flexibility
- Improved communication
- Reduce Maintenance cost
- Errors can be detected much earlier as the system is made side by side

DISADVANTAGES OF PROTOTYPE MODEL

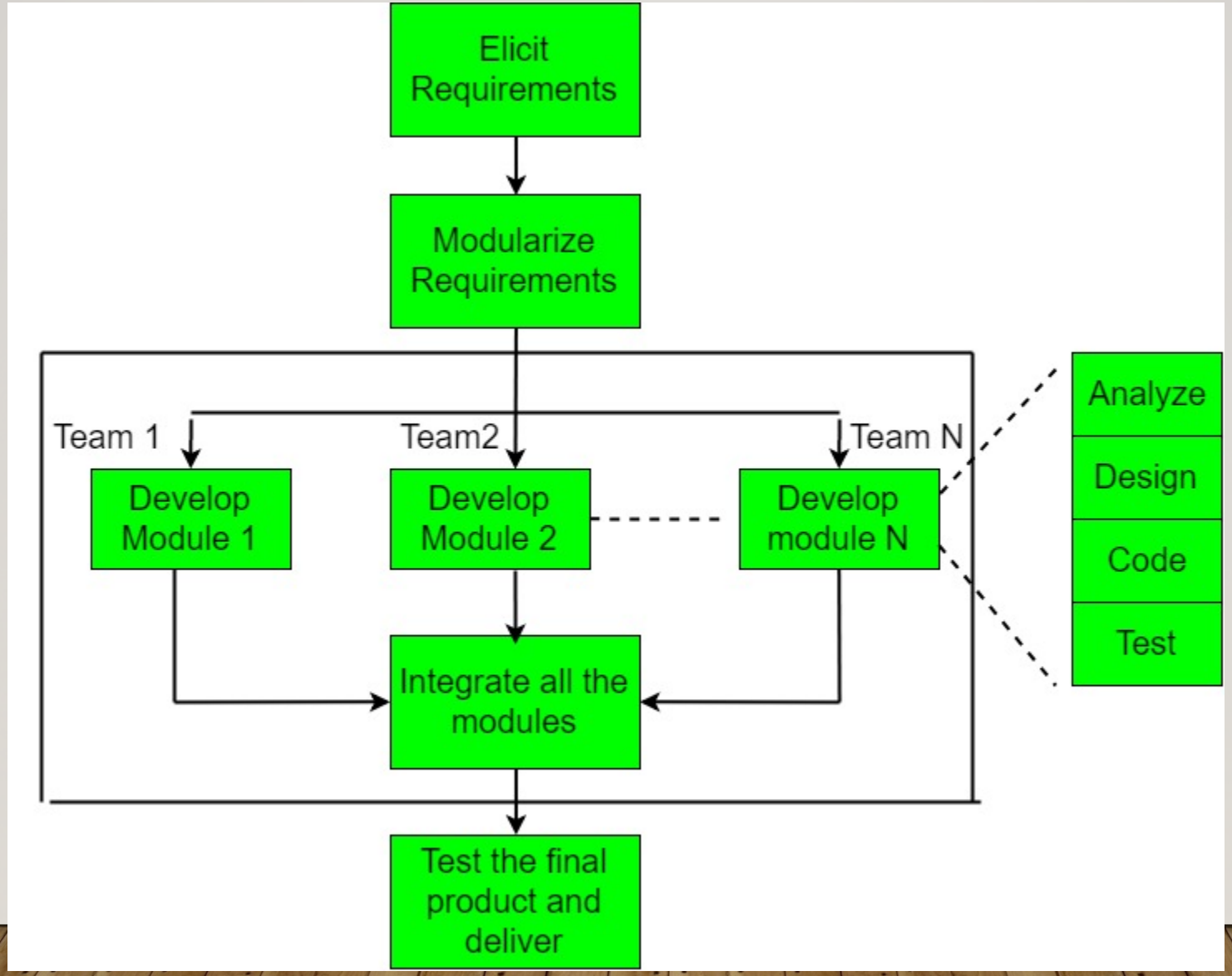
- Can be time-consuming and resource-intensive, especially if multiple iterations are needed.
- Requires strong project management to ensure clear goals and focus in each iteration.
- May not be suitable for highly complex or critical systems.
- Require extensive customer collaboration
  - ➢ Costs customer money
  - ➢ Needs committed customer
  - ➢ Difficult to finish if customer withdraw
  - ➢ May be too customer specific, no broad market

WHEN TO USE THE PROTOTYPE MODEL:

- When the requirements are not well-defined or are likely to change.
- When user feedback is crucial for shaping the product.
- When rapid delivery of a basic product is important.
- For projects with high uncertainty or complexity.

Overall, the prototype model is a valuable approach for software development, especially when dealing with uncertainty and the need for user feedback.

RAD (Rapid Application Development) MODEL:



Contd.

- The **RAD (Rapid Application Development)** model is a type of incremental model for software development.
- It emphasizes **quick development and iteration of prototypes** over rigorous planning and testing.
- **The RAD model is particularly suited for projects where requirements are expected to change frequently.**
  - ➢ Requirements Gathering
  - ➢ Rapid Prototyping
  - ➢ Testing and Feedback
  - ➢ Iteration and Development
  - ➢ Deployment and Delivery

ADVANTAGES OF RAD MODEL:

- The use of reusable components helps to reduce the cycle time of the project.
- Feedback from the customer is available at the initial stages.
- Reduced costs as fewer developers are required.
- The use of powerful development tools results in better quality products in comparatively shorter time spans.
- The progress and development of the project can be measured through the various stages.
- It is easier to accommodate changing requirements due to the short iteration time spans.
- Productivity may be quickly boosted with a lower number of employees.

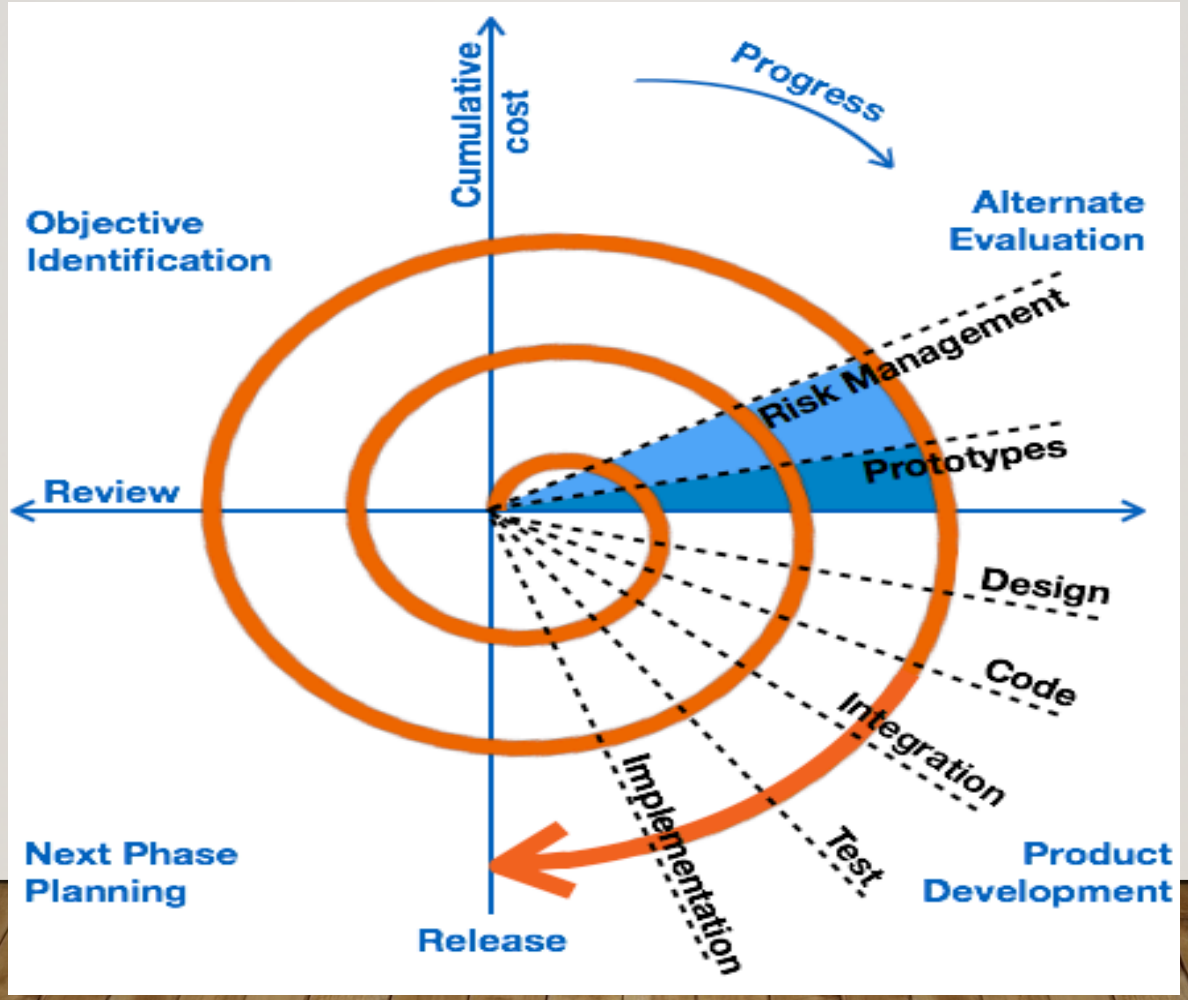
DISADVANTAGES OF RAD MODEL:

- Dependency on User Involvement
- Not Suitable for Large Projects
- Requires Skilled Developers
- Lack of Emphasis on Planning and Documentation
- Inadequate for Critical Systems
- Difficulty in Managing Changes
- Potential for Scope Creep
- Higher Cost of Software

WHEN TO USE RAD MODEL….

- When **the user is involved throughout the life cycle, the project can be time-boxed, the functionality delivered in increments, high performance is not required, low technical risks are involved and the system can be modularized.**
- When it is necessary to design a system that can be divided into smaller units within two to three months.
- When there is enough money in the budget to pay for both the expense of automated tools for code creation and designers for modeling.

SPIRAL MODEL



Contd.

- The spiral model is a **risk-driven software development process model** that **combines the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.** Imagine it as a staircase that spirals upwards, with each loop encompassing a phase of the development process.

  - **Planning:** In this initial phase, the project scope is defined, risks are identified, and a plan is created for the next iteration of the spiral.
  - **Risk Assessment:** Potential risks associated with the project are evaluated and prioritized. The focus is on mitigating high-risk factors before they cause major problems.
  - **Engineering:** Based on the identified risks and priorities, the software is developed in smaller, incremental releases. This allows for early feedback and adjustments to address potential issues.
  - **Evaluation:** The developed features are evaluated and tested, and feedback is incorporated into the next iteration.

SPIRAL MODEL IS ALSO CALLED **META MODEL**…..

- The Spiral model is called a Meta-Model because it **subsumes all the other SDLC models.** For example, a **single loop spiral actually represents the Iterative Waterfall Model**.
- The spiral model incorporates the **stepwise approach of the Classical Waterfall Model**.
- The spiral model uses the approach of the **Prototyping Model by building a prototype** at the start of each phase as a risk-handling technique.
- Also, the **spiral model can be considered as supporting the Evolutionary model** – the iterations along the spiral can be considered as evolutionary levels through which the complete system is built.

ADVANTAGES OF SPIRAL MODEL

- Risk Handling
- Good for large projects
- Flexibility in Requirements
- Customer Satisfaction
- Iterative and Incremental Approach
- Emphasis on Risk Management
- Improved Communication
- Improved Quality

DISADVANTAGES OF SPIRAL MODEL

- Complex
- Expensive
- Too much dependability on Risk Analysis: Without very highly experienced experts, it is going to be a failure to develop a project using this model.
- Difficulty in time management
- Time-Consuming
- Resource Intensive: The Spiral Model can be resource-intensive, as it requires a significant investment in planning, risk analysis, and evaluations.

WHEN TO USE SPIRAL MODEL….

- Uncertain or Evolving Requirements
- Large and Complex Projects
- High-Risk Projects
- Projects Requiring Prototyping
- Projects with Incremental Releases
- Projects with Active Customer Involvement
- Phased and Controlled Development
- Projects Requiring Transparent Progress Tracking

REQUIREMENT ENGINEERING

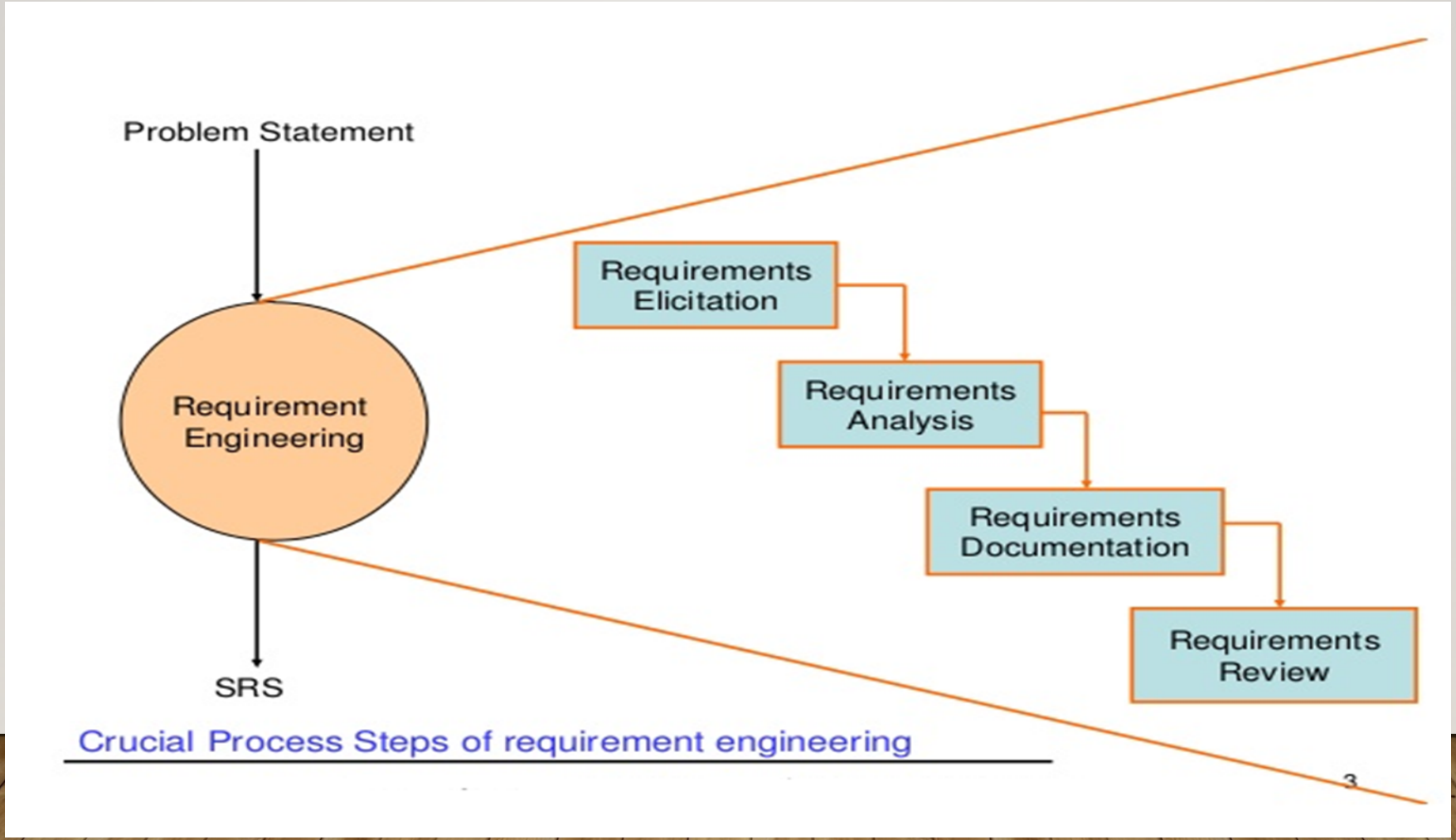
- **Requirements describe : What? not How?**
- It produces one large document written in natural language contains a description of what the system will do without describing how it will do it. **The input to requirement engineering is the problem statement prepared by the customer**.
- Without well written document

➢ -- Developers do not know what to build

➢ -- Customers do not know what to expect

➢ -- There is no way to validate that the built system satisfies the requirements

PROCESS STEPS OF REQUIREMENT ENGINEERING



Contd.

- **Requirements Elicitation:** This is also known as gathering of requirements. Here requirements are identified with the help of customer and existing systems processes.
- **Requirements Analysis:** The requirements are analyzed in order to identify inconsistencies, defects, omissions etc. We describe requirements in terms of relationships and also resolve conflicts, if any.
- **Requirements Documentation:** It is end product of requirements elicitation and analysis. The documentation is very important as it will be the foundation for the design of the software. The document is known as SRS. SRS may act as a contract between developer and customer.
- **Requirements Review:** The review process is carried out to improve the quality of the SRS. It may also be called as requirements verification

CHALLENGES TO REQUIREMENT ENGINEERING

- Requirements are difficult to uncover
- Requirements change
- Tight project Schedule
- Communication barriers
- Market driven software development
- Lack of resources

TYPES OF REQUIREMENTS:

- **Functional Requirement:** These are the backbone of any software system, representing the specific actions and outcomes it must deliver. **They describe what the system should do rather than how it should do it**.

- **Non- Functional Requirement:** As the name suggests, **specify how a system should perform rather than what it should do**. These are essentially the quality attributes of a system, **describing how well it operates and meets user expectations**. It's focus is on the overall experience and performance.

FEASIBILITY STUDY

- A Feasibility Study is a study made before committing to a project. It leads to a decision:
- -- go ahead
- -- do not go ahead
- -- think again
- The main purpose of the feasibility study is to answer one main question: **“Should we proceed with this project idea?”**
- It concentrates on the following area :-

➢ **Operation Feasibility**

➢ **Technical Feasibility**

➢ **Economic Feasibility**

PURPOSE OF FEASIBILITY STUDIES

- The decision to implement any new project or program must be based on a thorough analysis of the proposed project/program.
- A feasibility study is defined as an evaluation or analysis of the potential impact of a proposed project or program.
- **It is conducted to assist decision makers in determining whether or not to implement a project program.**
- It should also **contain analysis related to financial and operational impact** and should include advantages and disadvantages of both the current situation and the proposed project.

FOCUS OF FEASIBILITY STUDY

- Is the project concept feasible?
- Will it be possible to develop a product that matches the project's vision statement?
- What are the current estimated cost and schedule for the project?
- How big is the gap between original cost and current estimates?
- Have the major risks to the project been identified?
- Are the specifications complete and stable enough to support remaining development work?
- Have users and developers are agree for user interface prototype?
- Is the s/w development plan is complete to support further development?

REQUIREMENTS ELICITATION

- It is the **Most difficult, Most critical, Most error prone and Most communication intensive aspect of s/w development**.
- Elicitation can succeed only through an effective customer developer partnership.
- **Requirements are gathered by asking questions, writing down the answers etc**. hence, it is most communication intensive activity of software development.
- Developers and users have different mind set, expertise and vocabularies. Due to communication gap, there are chances of inconsistencies, misunderstanding and omission of requirements.

REQUIREMENT ELICITATION METHODS:

- Interviews
- Brainstorming Sessions
- Facilitated Application Specification Technique (FAST)
- Quality Function Deployment
- The Use Case Approach

INTERVIEWS

- After receiving the problem statement from the customer, **the first step is to arrange a meeting with the customer**. During meeting or interview, both the parties would like to understand each other.
- Both the parties have different feelings, goals, opinions, vocabularies, understandings, but one thing is common, both wants the project to be a success.
- Normally specialized developers called **'requirement engineers' interact with the customer.**
- **Interview may be open-ended or structured.**
- In **open-ended there will be pre-set agenda**. Context free questions may be asked to understand the problem.
- In **structured interview, agenda of fairly open questions is prepared**. Sometimes a proper questionnaire is designed for the interview.

SELECTION OF STAKEHOLDER:

- **Entry level personnel:** They **may not have sufficient domain knowledge and experience**
- **Middle level stakeholder:** They **have better domain knowledge and experience of the project.** They know the sensitive, complex and critical areas of the project.
- **Managers**: Higher level of **management officers like vice president, general managers, managing directors should also be interviewed**.
- **Users of the software:** This group is most important because **they will spend more time interacting with the software than any one else**. Their information may be eye opener and may be original.

TYPES OF QUESTIONS

- Are there any problems with existing system?
- Have you faced any calculation errors in past?
- What are the possible reasons for malfunctioning?
- How many students are Enrolled?
- What are possible benefits of computerizing this system?
- How are you maintaining the records of previous students?
- Any requirement of data from other system?
- Any specific problems?
- Any additional functionality?

BRAINSTORMING SESSIONS

- The **group discussion** may lead to new ideas quickly and help to promote creative thinking.
- It **promotes creative thinking, generates new ideas and provides platform to share views, expectations and difficulties of implementation**.
- All participants are encouraged to say whatever ideas come to mind, whether they seems relevant or not.
- **This group technique may be carried out with specialized groups like actual users, middle level managers and total stakeholders**.

FACILITATED APPLICATION SPECIFICATION TECHNIQUES (FAST)

- **Objective is to bridge the expectation gap:** A difference between what developers think they are supposed to build and what customers think they are going to get.

➢-- Similar to brainstorming sessions.

➢-- Team oriented approach

➢-- Creation of joint team of customers and developers.

- **Each attendee is asked to make a list of objects that are:**

➢**--Part of the environment that surrounds the system.**

➢**--Produced by the system.**

➢**--Used by the system.**

Contd.

- Each participant prepares his/her list
- Different lists are then combined
- Redundant entries are eliminated
- Team is divided into smaller sub-teams to develop mini-specifications
- And finally a draft of specifications is written down using all the inputs from the meeting

QUALITY FUNCTION DEPLOYMENT(QFD)

- In QFD, **Prime concern is customer satisfaction and what is valuable to the customer.**
- **Types of Requirements**
- --Normal Requirements: Example – normal requirements for a result management system may be entry of marks, calculation of results, etc.
- --Expected Requirements: Example – protection from unauthorized access.
- --Exciting requirements: Example – when unauthorized access is detected, it should backup and shutdown all processes.

Contd.

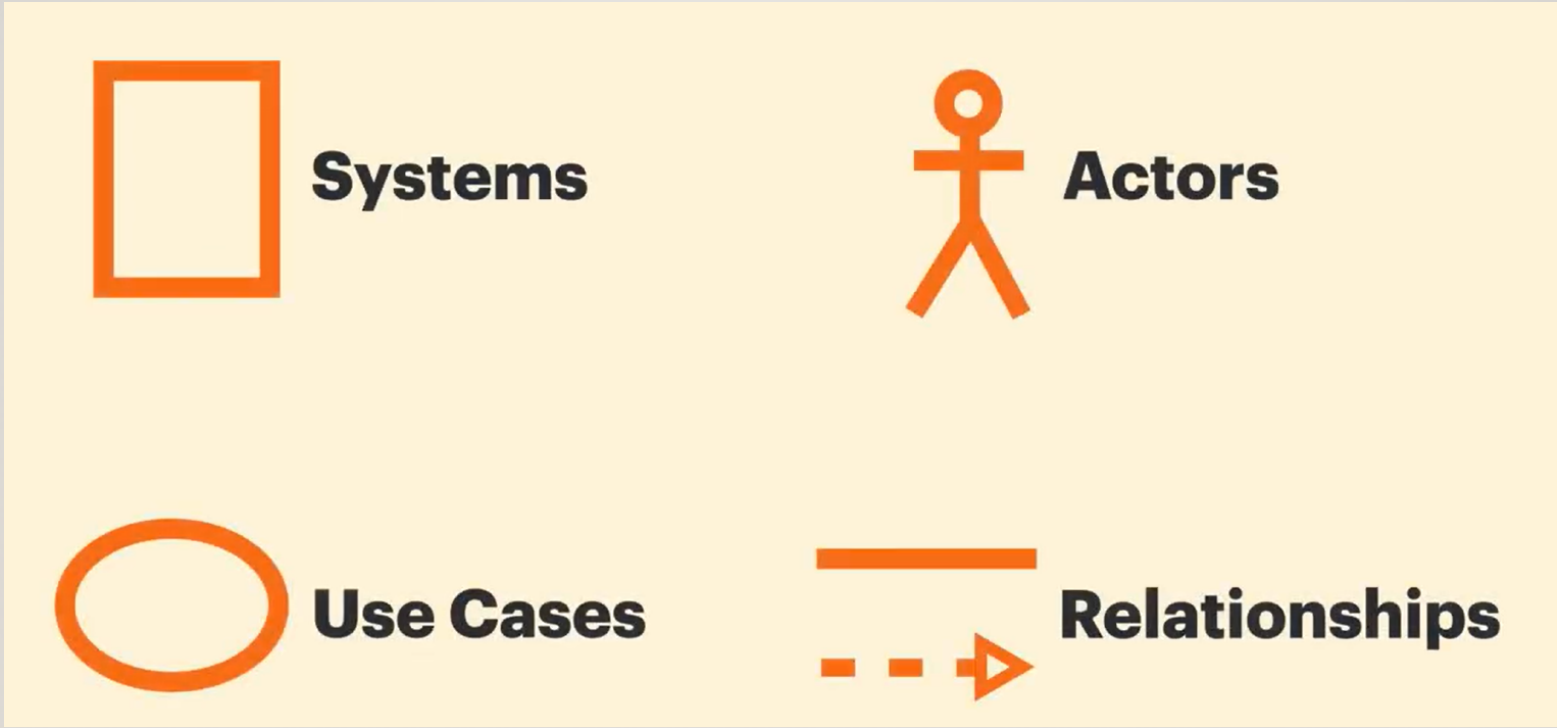
- In the Quality Function Deployment (QFD) process, **a common practice is to assign points or weights to each requirement to indicate their relative importance**.
- This helps in prioritizing requirements and focusing resources on the most critical aspects of the project.
- **The assignment of points or weights is typically done during the development of the House of Quality (HOQ), which is a central tool in the QFD methodology.**

5	Points	:	V. Important
4	Points	:	Important
3	Points	:	Not Important but nice to have
2	Points	:	Not important
1	Points	:	Unrealistic, required further exploration

USE CASE APPROACH

- The Use-case model is defined as **a model which is used to show how users interact with the system in order to solve a problem**.
- It defines the user's objective, **the interactions between the system and the user, and the system's behavior required to meet these objectives.**
- The use-case model is used like the main specification of the system functional requirements as the basis **for design and analysis,** as the basis **for user documentation,** as the basis **of defining test cases,** and **as an input to iteration planning.**
- A use case diagram is a type of **Unified Modeling Language (UML) diagram.**

COMPONENTS OF BASIC MODEL



Contd.

- **Actor:** Usually, actors are **people involved with the system defined on the basis of their roles**. An actor can be anything such as human or another external system.
- **Use Case:** The use case defines **how actors use a system to accomplish a specific objective.** The use cases are generally introduced by the user to meet the objectives of the activities and variants involved in the achievement of the goal.
- **Associations:** Associations are another component of the basic model. It is used to **define the associations among actors and use cases they contribute in**. This association is called **communicates-association**.

SYSTEM

- A system is the product, service, or software under discussion.
- It is whatever you are developing:

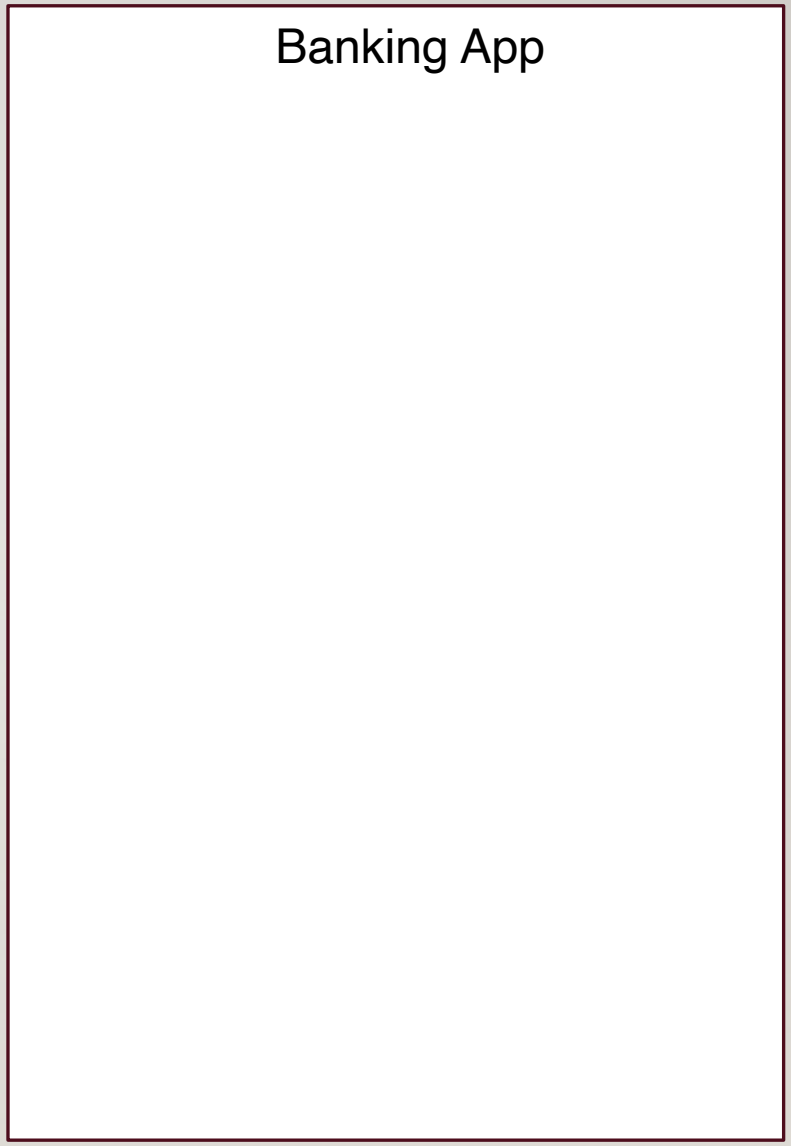
➢--Website

➢--Software Component

➢--Business Process

➢--App

- This rectangle defines the scope of a system.
- Anything that happens inside the system is enclosed within this rectangle.
- Anything outside this doesn't happen inside the banking app.



ACTOR

- Actor is something or someone that uses our system to achieve a goal.

➢--Person

➢--Organisation

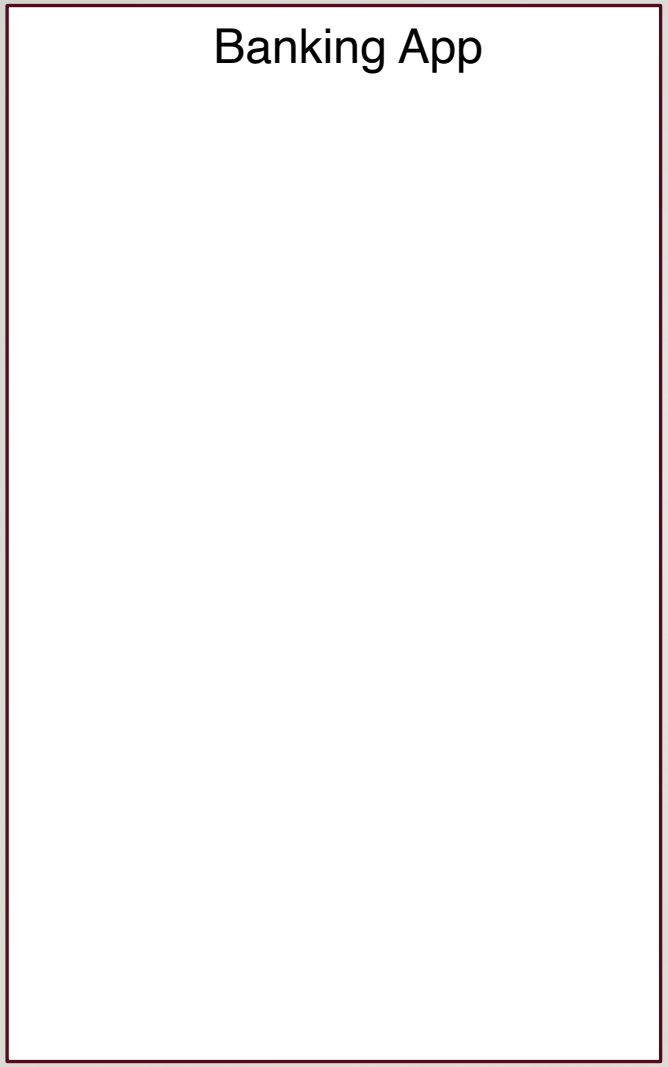
➢--Another System

➢--An External Device

- So, who or what is using our system is a actor.

Customer

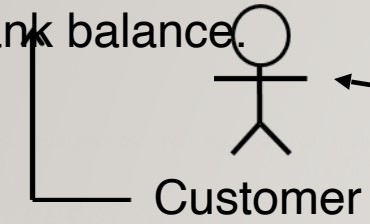
Bank



Things to Keep in Mind:

1. Actors are external objects and need to be placed outside the system.
2. Actors need to be thought of as types or categories.
3. Name of the actors should be categorical and not like Sam or X-Bank.

As they will initiate the use of system by opening the banking app and will perform so task with it, like checking there bank balance.

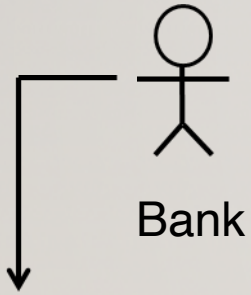


Customer

Banking App

Primary Actors

Initiates the use of the system.



Bank

Secondary Actors

Reactionary

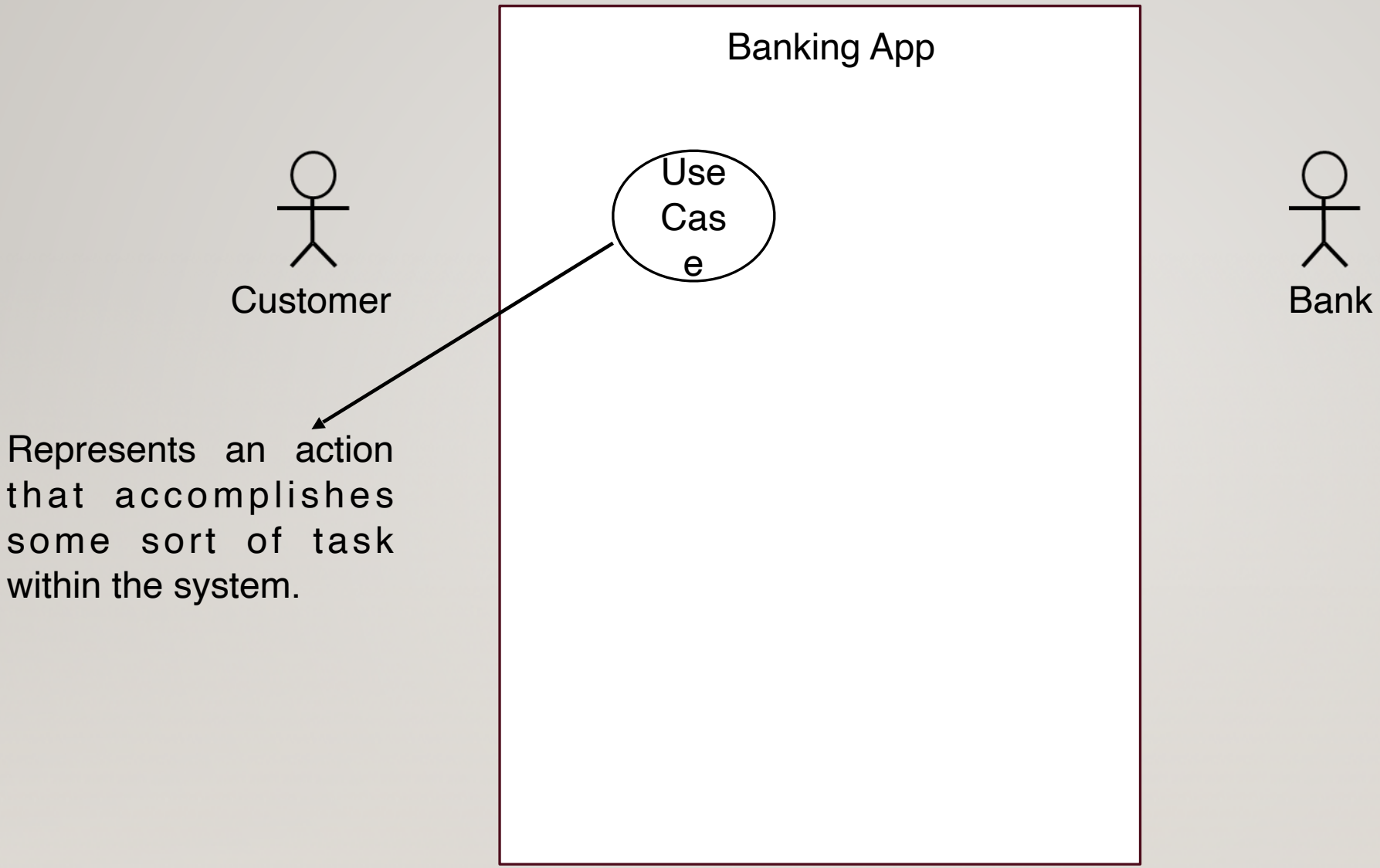
As the Bank will only going to act once the customer will do something.

Customer

Banking App

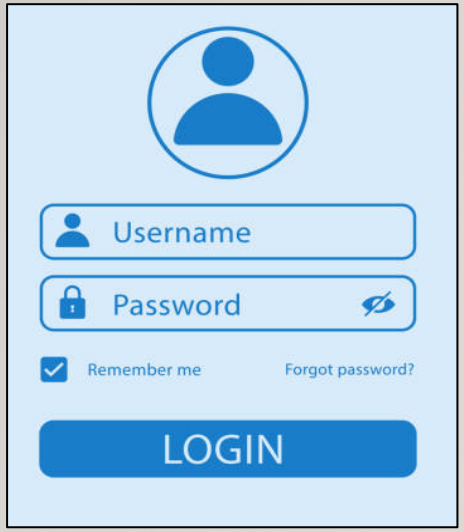
Bank

Primary Actors should be on left side whereas secondary actors should be on right side.



Represents an action that accomplishes some sort of task within the system.

WHAT WILL OUT BANKING APP WILL DO?



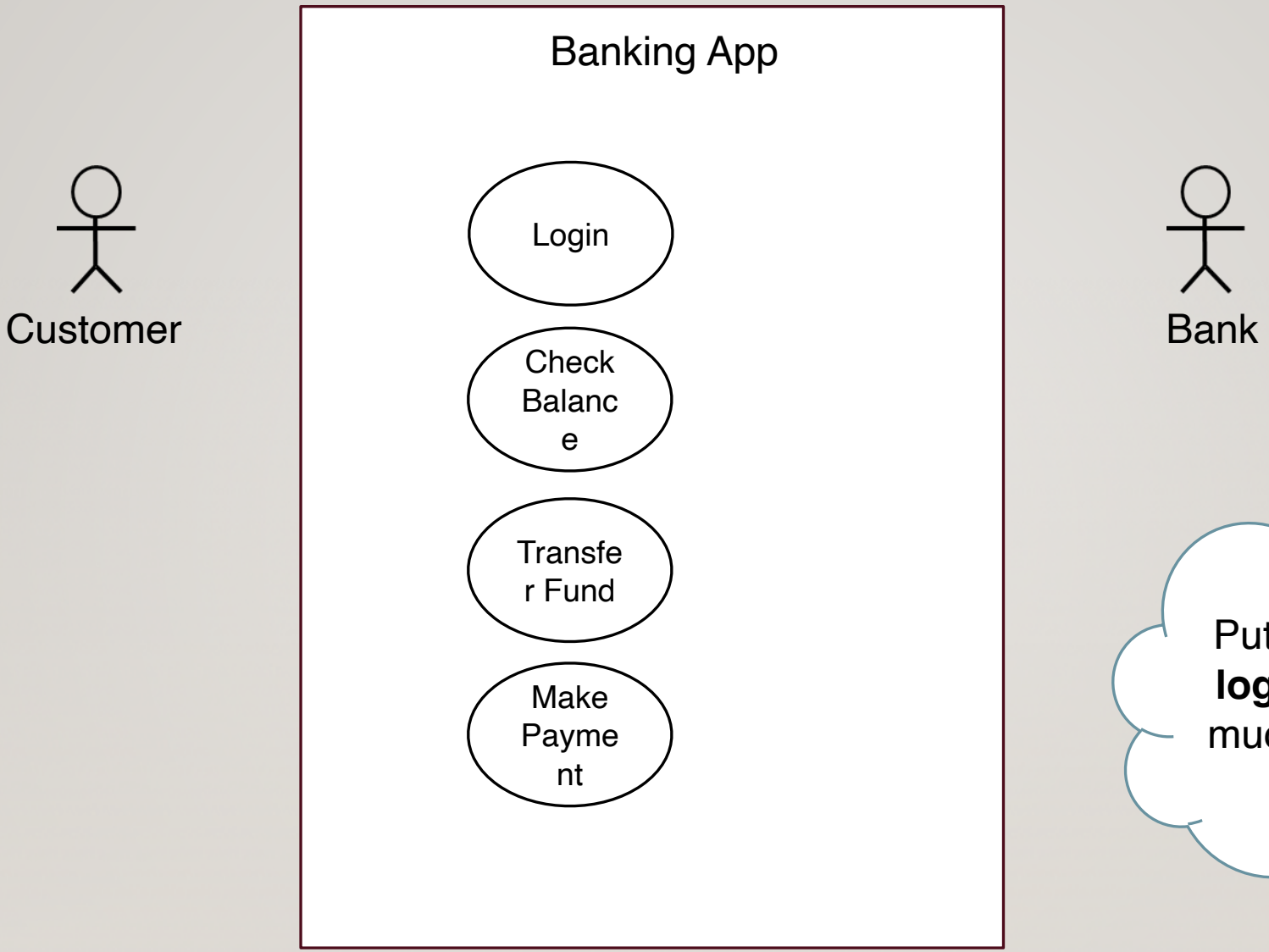
Allows User to login

Check Account Balance

Transfer Funds

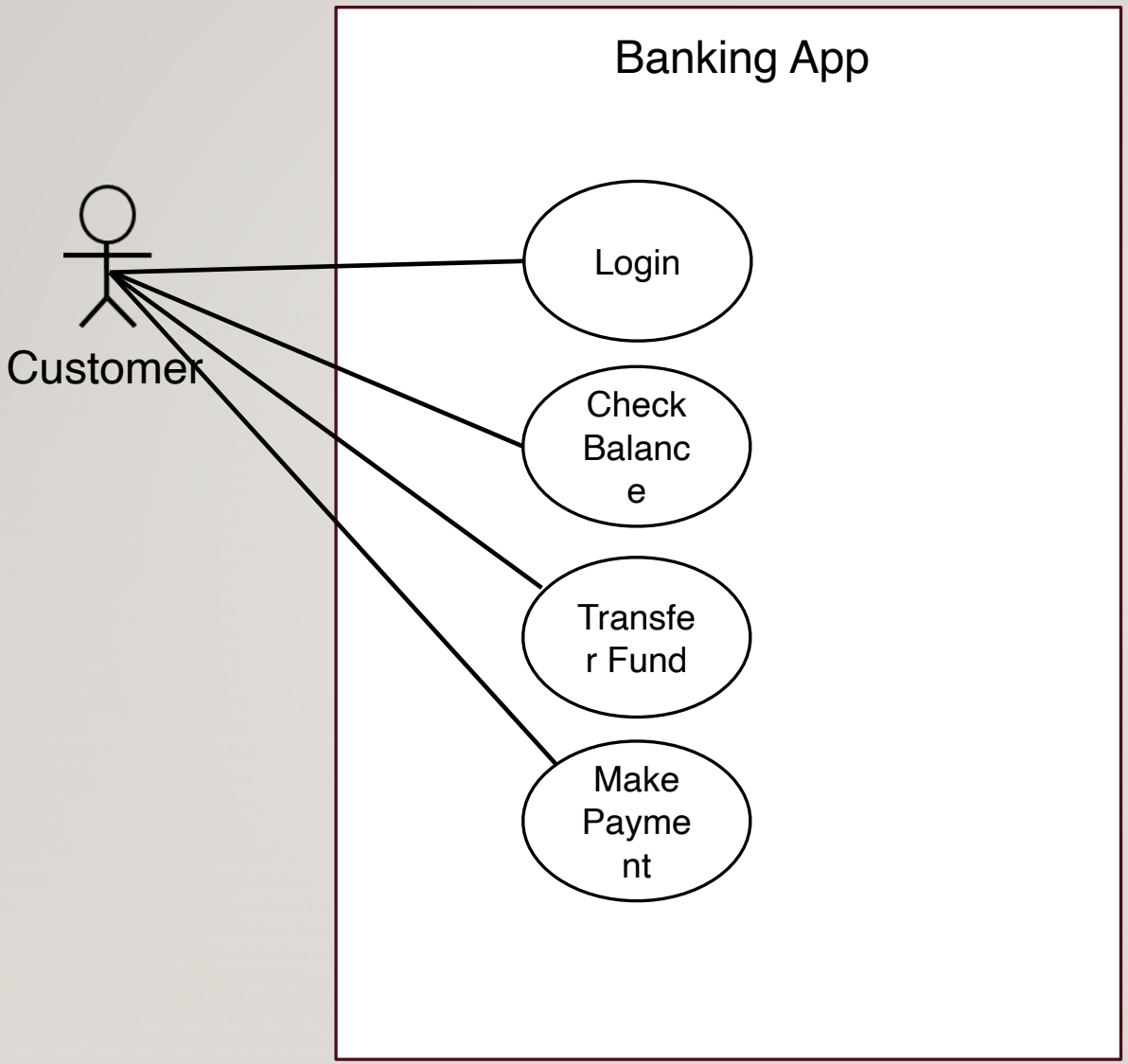
Pay Bills

So, if this is what our banking system does, then we should have use cases for all these functionalities.


Banking App
Login
Check Balanc e
Transfe r Fund
Make Payme nt
Customer
Bank
Put Use Cases in **logical order**, as much as possible.

RELATIONSHIPS

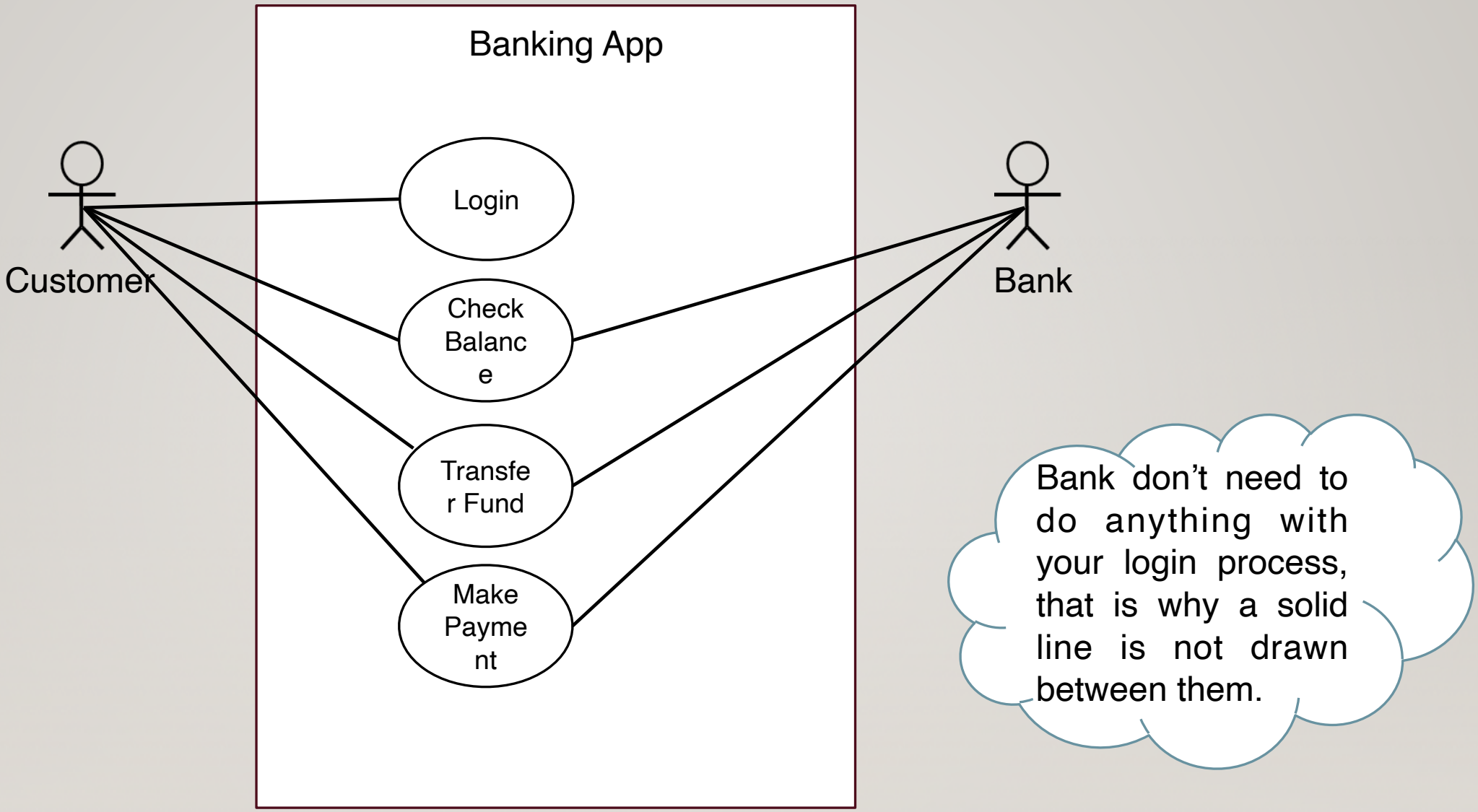
- The final element in use case diagram is relationships.
- Its act by a definition is using our system to achieve a goal.
- **Remember**, each actor has to interact with at least one of the use cases within our system.
- In our example, a customer is going to login with our banking app, so we draw a solid line between the actor and that particular use case to show the relationship.
- This type of relationship is called the **Association Relationship** and **it just signifies the basic communication for interaction.**
- Our customer will also check balance, transfer funds, make payment; so we draw a solid line between them as well.



Remember each actor has to interact with at least one use case, whether it is primary actor or secondary actor.

So,

- When the customer wants to check balance then bank is going to provide with the current bank balance.
- Similarly, when the customer wants to transfer funds or make a payment, then the bank will follow up.



Bank don't need to do anything with your login process, that is why a solid line is not drawn between them.

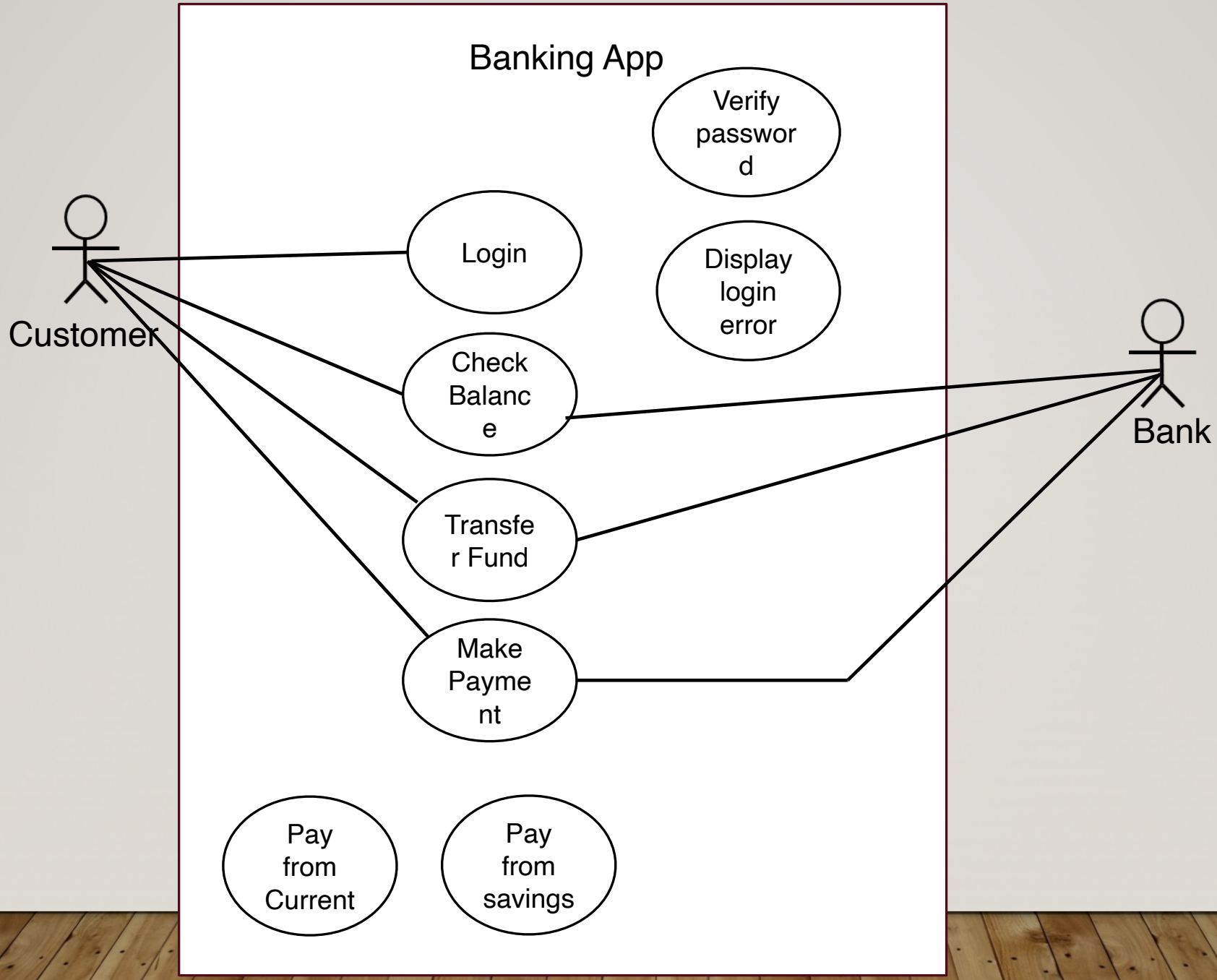
RELATIONSHIP TYPES

- Association
- Include
- Extend
- Generalization

When the user tries to login, then the banking app is going to verify the password before completing the login process

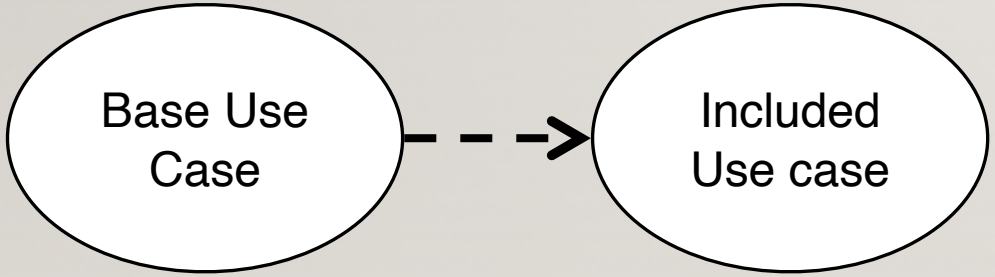
But if the password is incorrect

The banking app will show an error message.

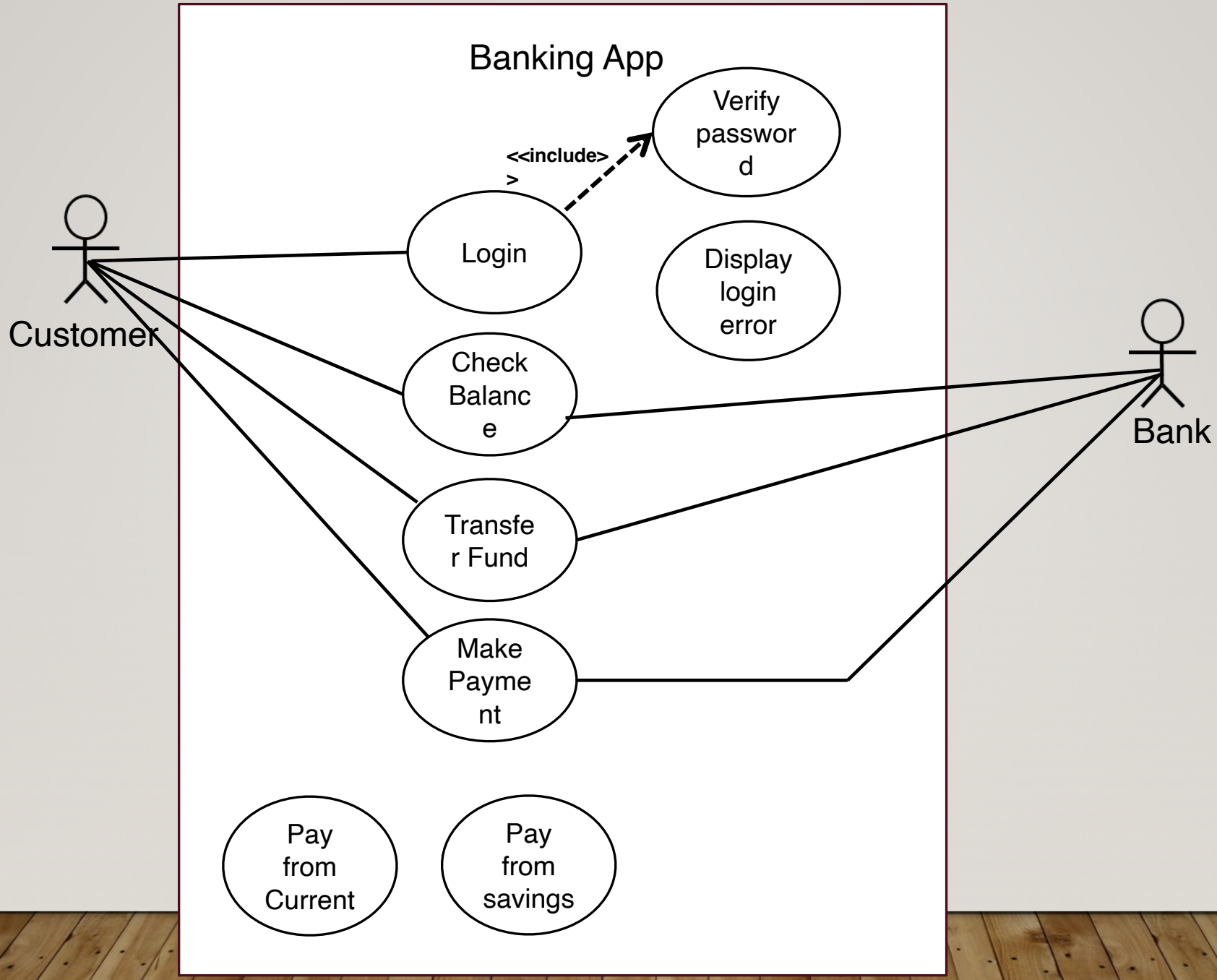

Banking App
Verify password
Login
Display login error
Customer
Check Balance
Bank
Transfer Fund
Make Payment
Pay from Current
Pay from savings

INCLUDE RELATIONSHIP

- How is verify password is related to our diagram? Neither of our actors are directly initiating this action. It is going to happen every time, when a customer is trying to login in our banking app.
- Such kind of relationships are own as **include relationships**.

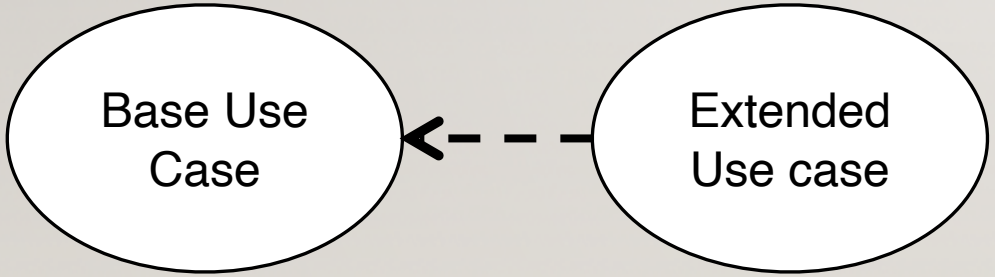


- It shows **dependency between** base use case and included use case.
- Every time a base use is executed, the included use case is executed as well.
- **Base use case requires included use case in order to get completed**.

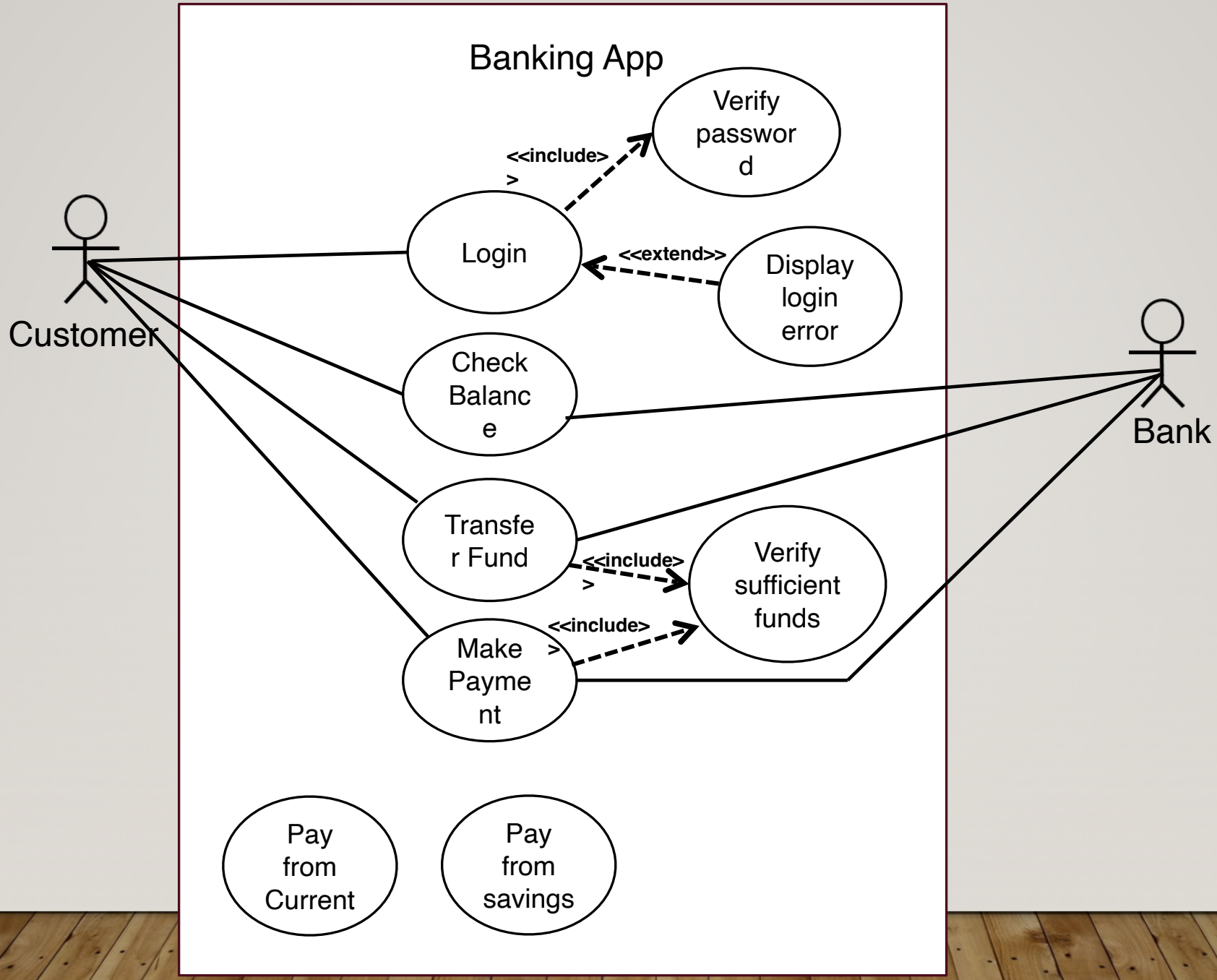

Banking App
Verify password
<<include>>
Login
Display login error
Customer
Check Balance
Bank
Transfer Fund
Make Payment
Pay from Current
Pay from savings

EXTEND RELATIONSHIP

- When **base use case is executed, the extend use case will happen sometimes** but not every time. **It will only happen when certain criteria are matched.**
- Here, **you have the option to extend the behaviour of base use case.**

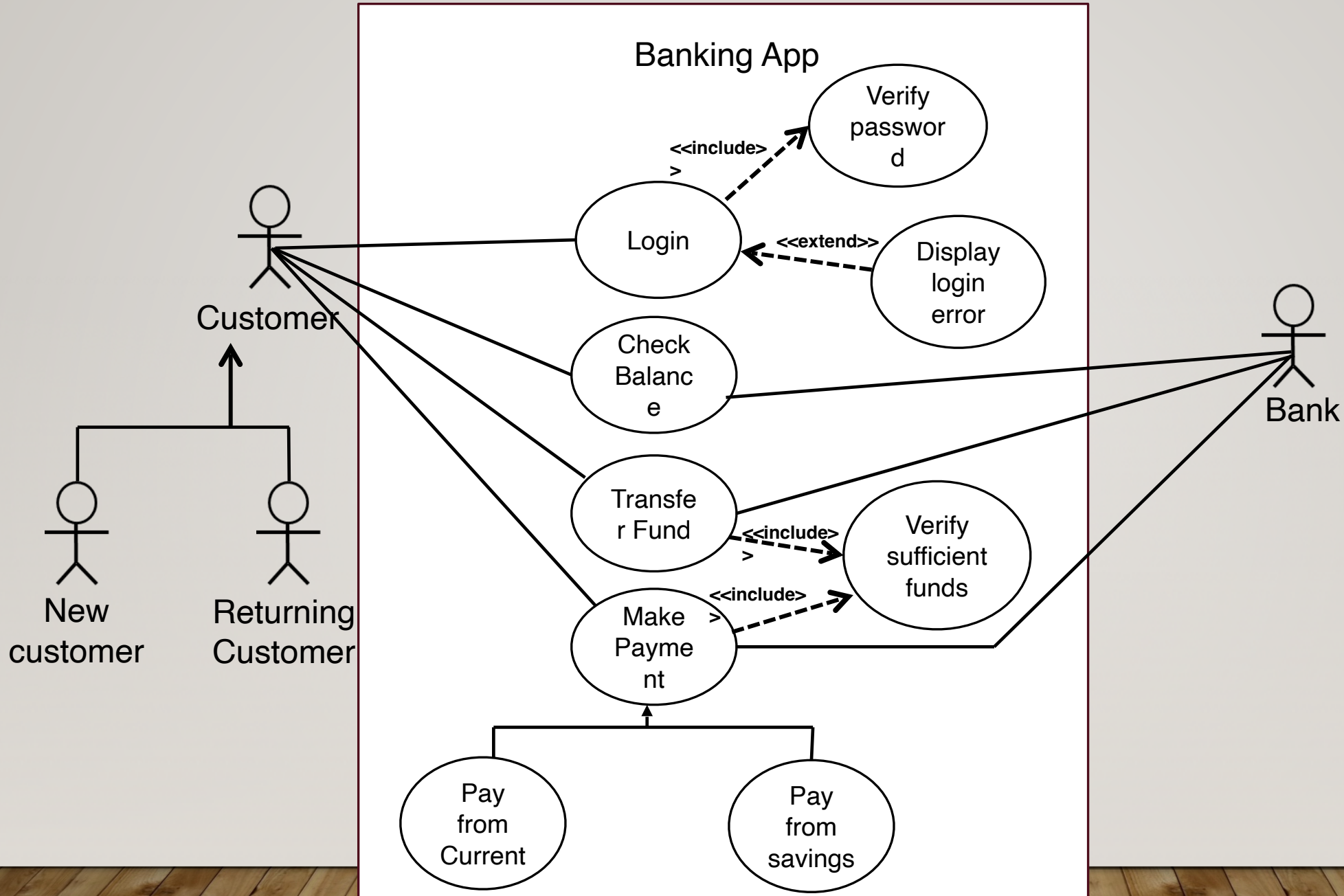


- In our example **login is a base use case, while display login error is an extend use case.**


Banking App
Verify password
<<include>>
Login
<<extend>>
Display login error
Customer
Check Balance
Bank
Transfer Fund
<<include>>
Verify sufficient funds
<<include>>
Make Payment
Pay from Current
Pay from savings

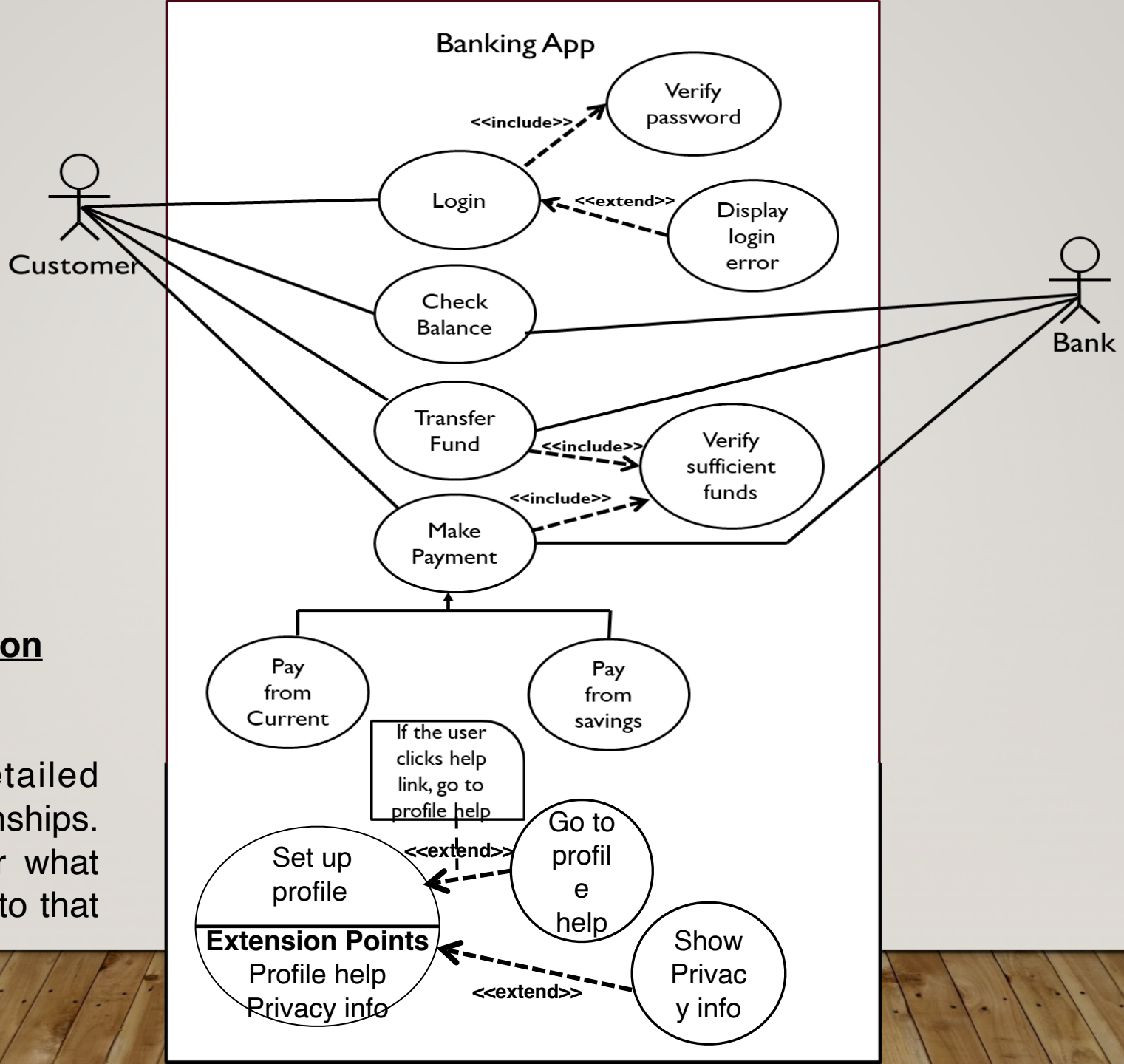
GENERALIZATION RELATIONSHIPS

- Also known as **inheritance**.
- When you make a payment, you can do it by a current account or from a savings account.
- In this scenario, **make payment is a general use case, while pay from current and pay from savings is are specialized use case**. (You can also say it as parent/child).
- You can have generalized relationships with actors as well.

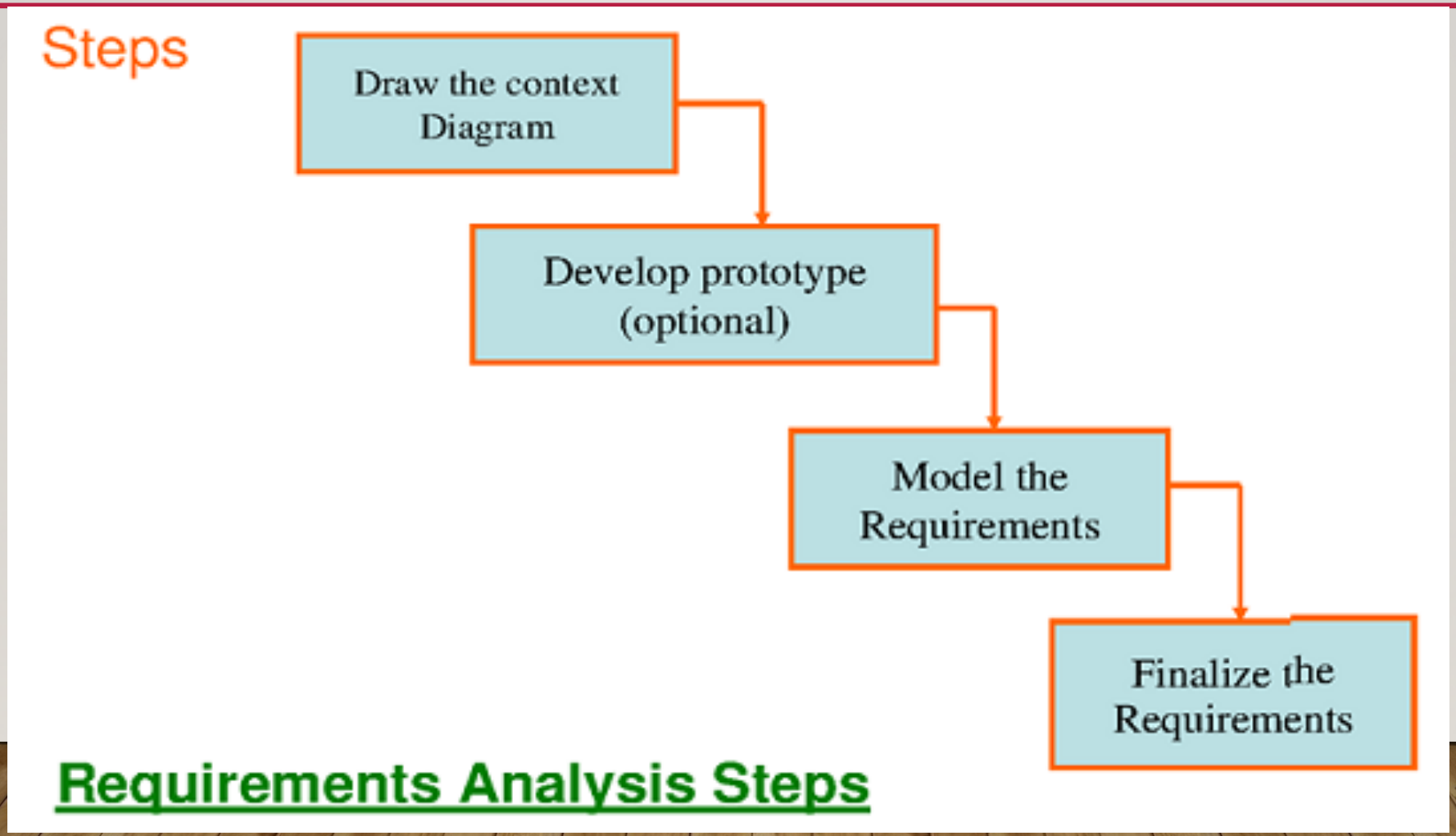

Banking App
Verify password
<<include>>
Login
<<extend>>
Display login error
Customer
Check Balance
Bank
Transfer Fund
<<include>>
Verify sufficient funds
<<include>>
Make Payment
New customer
Returning Customer
Pay from Current
Pay from savings

Use case with Extension Points

These are just the detailed version of extend relationships. We can also add note for what sort of conditions will lead to that extension point

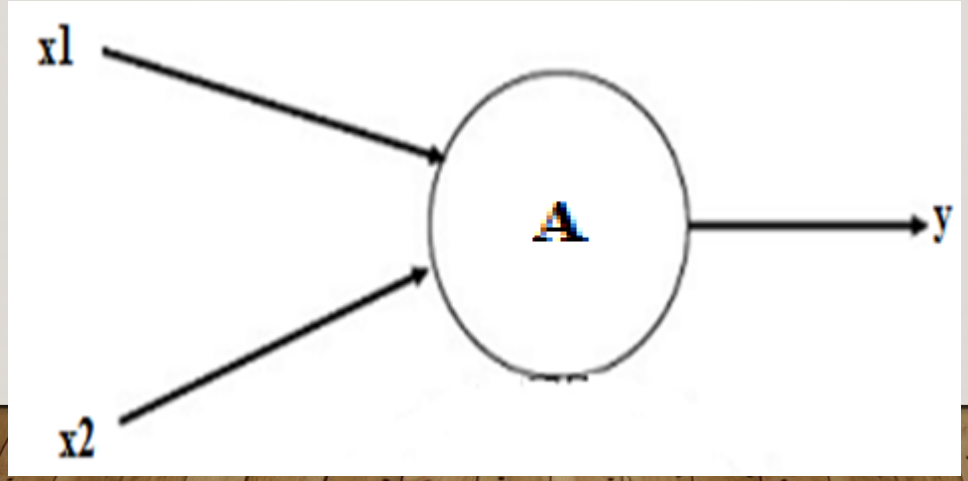


REQUIREMENT ANALYSIS



CONTEXT DIAGRAM (LEVEL 0-DATA FLOW DIAGRAM)

- This is also called as **Fundamental System Model** or **Level-0 DFD**.
- It represents the entire system element as a Single Bubble with input and output data indicated by incoming and outgoing arrows.
- **A context diagram provides a general overview of a process, focusing on its interaction with outside elements rather than its internal sub-processes.**



MODEL THE REQUIREMENTS

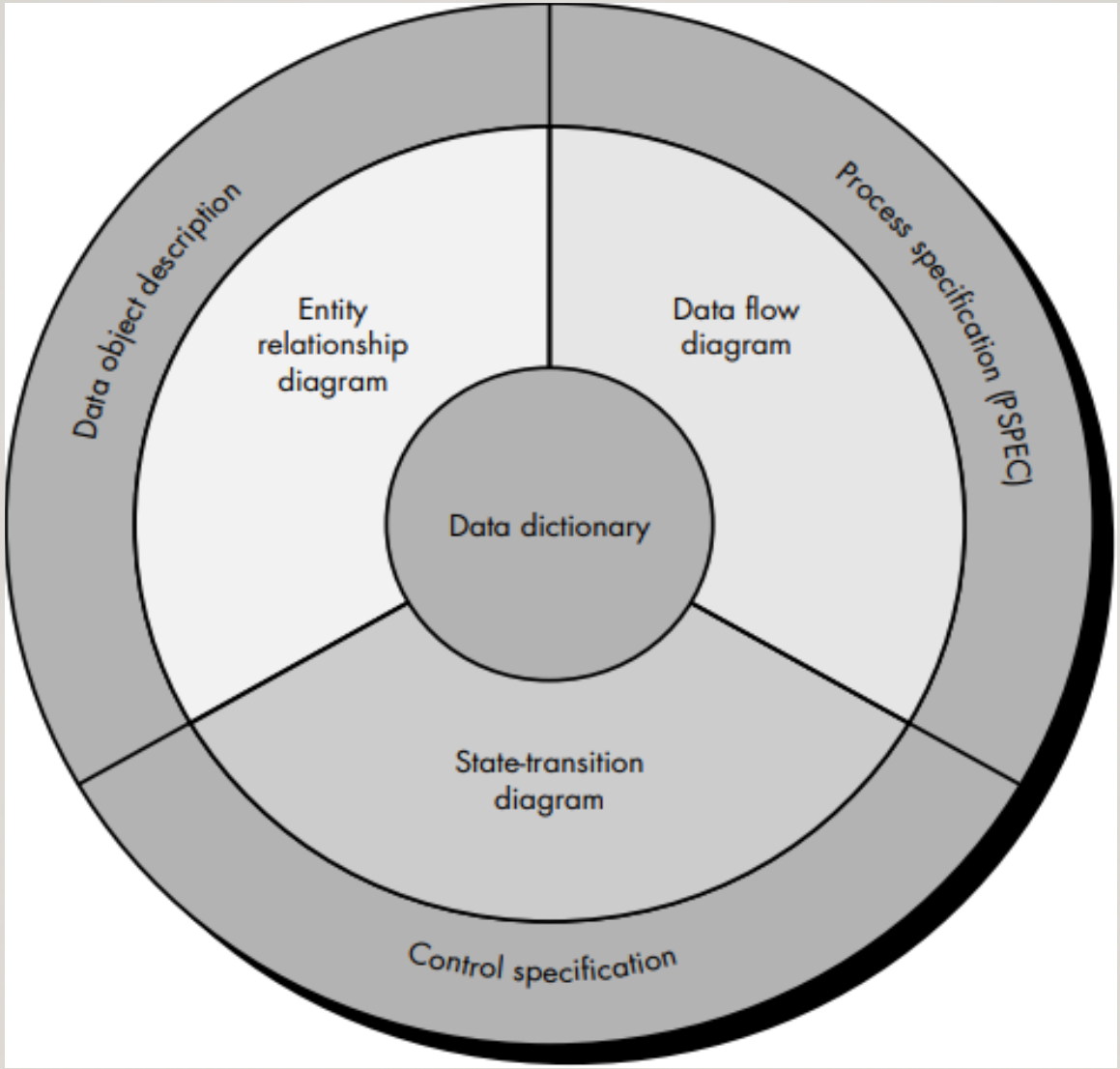
- These models are often referred as **analysis model.**
- The analysis model must achieve three primary objectives:

➢--To describe what the customer requires,

➢--To establish a basis for the creation of a software design, and

➢--To define a set of requirements that can be validated once the software is built.

The structure of the analysis model



DATA DICTIONARY

- At the core of the model lies the data dictionary—**a repository that contains descriptions of all data objects consumed or produced by the software**.

DATA FLOW DIAGRAM(DFD)

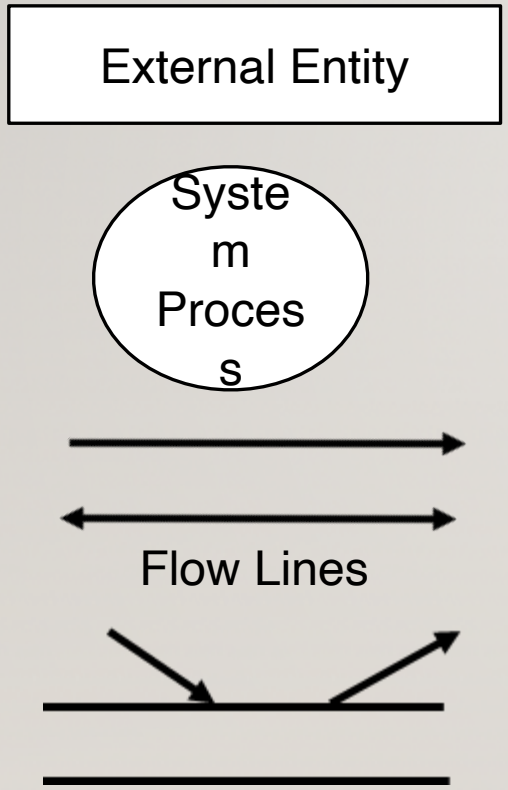
- They serves two purposes:

➢-- to provide an indication of **how data are transformed as they move through the system.**

➢--to **depict the functions (and subfunctions) that transform the data flow.**

- The DFD provides additional information that is used during the analysis of the information domain and serves as a basis for the modeling of function.
- **A description of each function presented in the DFD is contained in a process specification (PSPEC).**

SYMBOLS USED IN DATA FLOW DIAGRAMS

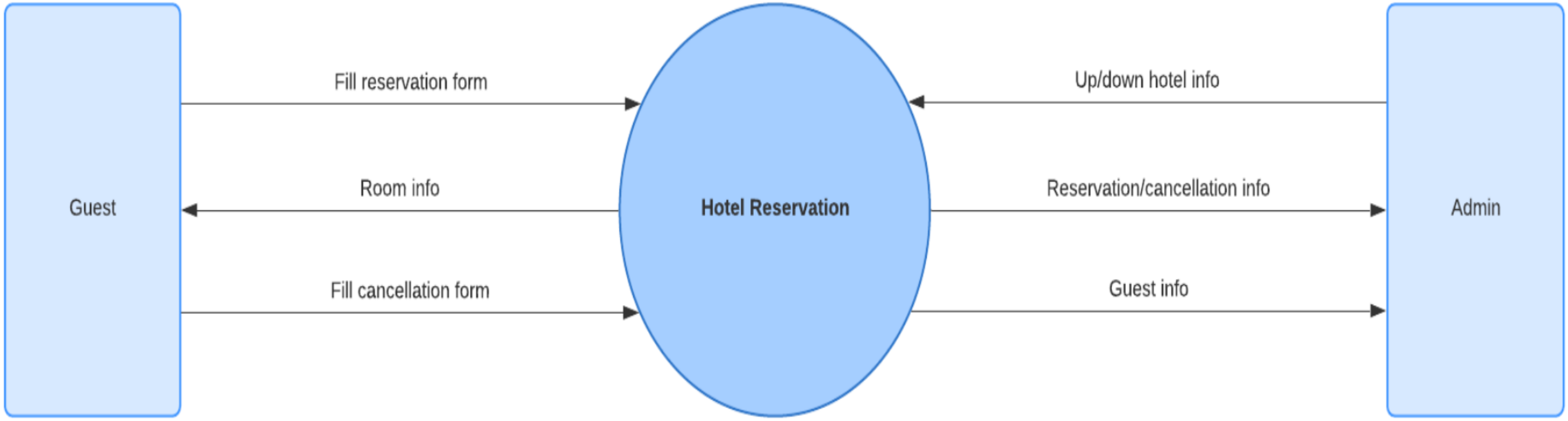


- **External Entity**- An element in the system diagram that inputs data into the information system and retrieves processed data.
- **Process**- Refers to the entire process of the system. This is responsible for processing and distributing information to the entities of the system context diagram.
- **Flow Line**- This element depicts the flow of the data within the system. It is supported by text to show what type of data is being sent.
- **Data Store**- A repository of data, the arrowheads indicate net inputs and net outputs to store.

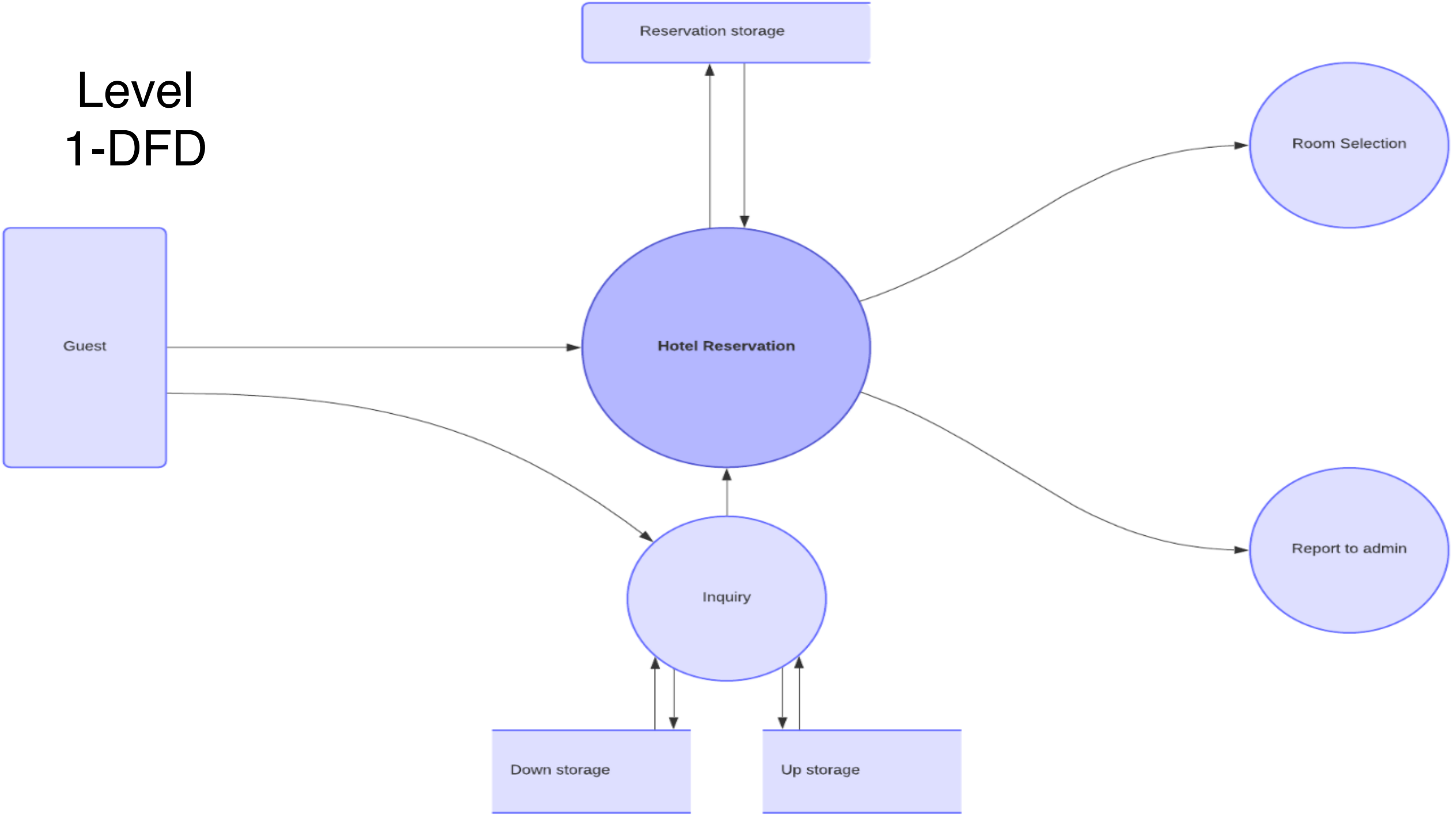
LEVELS IN DATA FLOW DIAGRAMS(DFD)

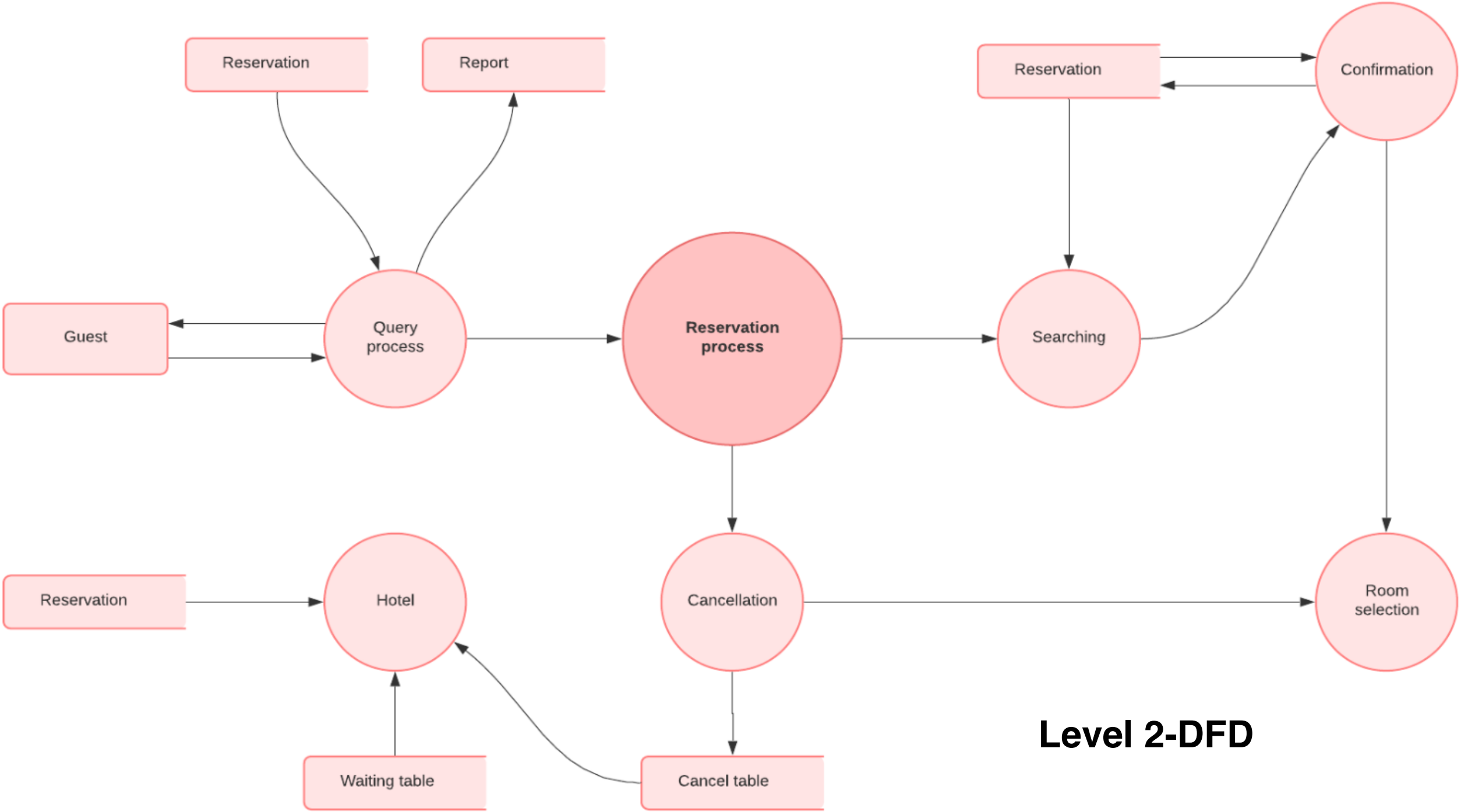
- **Level 0 DFD:** This provides **an overview of the entire system**. It shows the major processes, data flows, and data stores in the system, **without providing any details about the internal workings of these processes**.
- **Level 1 DFD:** This provides **more detailed view of the system by breaking down the major processes identified in the level 0 DFD into sub-processes**. Each sub-process is depicted as a separate process on the level 1 DFD.
- **Level 2 DFD:** This provides a detailed view of the system by breaking down the sub-processes into further sub-processes. **Each sub-process is depicted as a separate process on the level 2 DFD.**
- **Level 3 DFD:** This is the **most detailed level of DFDs**, which provides a detailed view of the processes, data flows, and data stores in the system. **This level is typically used for complex systems, where a high level of detail is required to understand the system. Each process on the level 3 DFD is depicted with a detailed description of its input, processing, and output.**

LEVEL 0-DFD FOR HOTEL RESERVATION



Level 1-DFD


Reservation storage
Room Selection
Guest
Hotel Reservation
Report to admin
Inquiry
Down storage
Up storage


Reservation
Report
Reservation
Confirmation
Guest
Query process
Reservation process
Searching
Reservation
Hotel
Cancellation
Room selection
Waiting table
Cancel table

Level 2-DFD

POINTS TO REMEMBER:

- DFD show the flow of data through the system.

➢ --All names should be unique

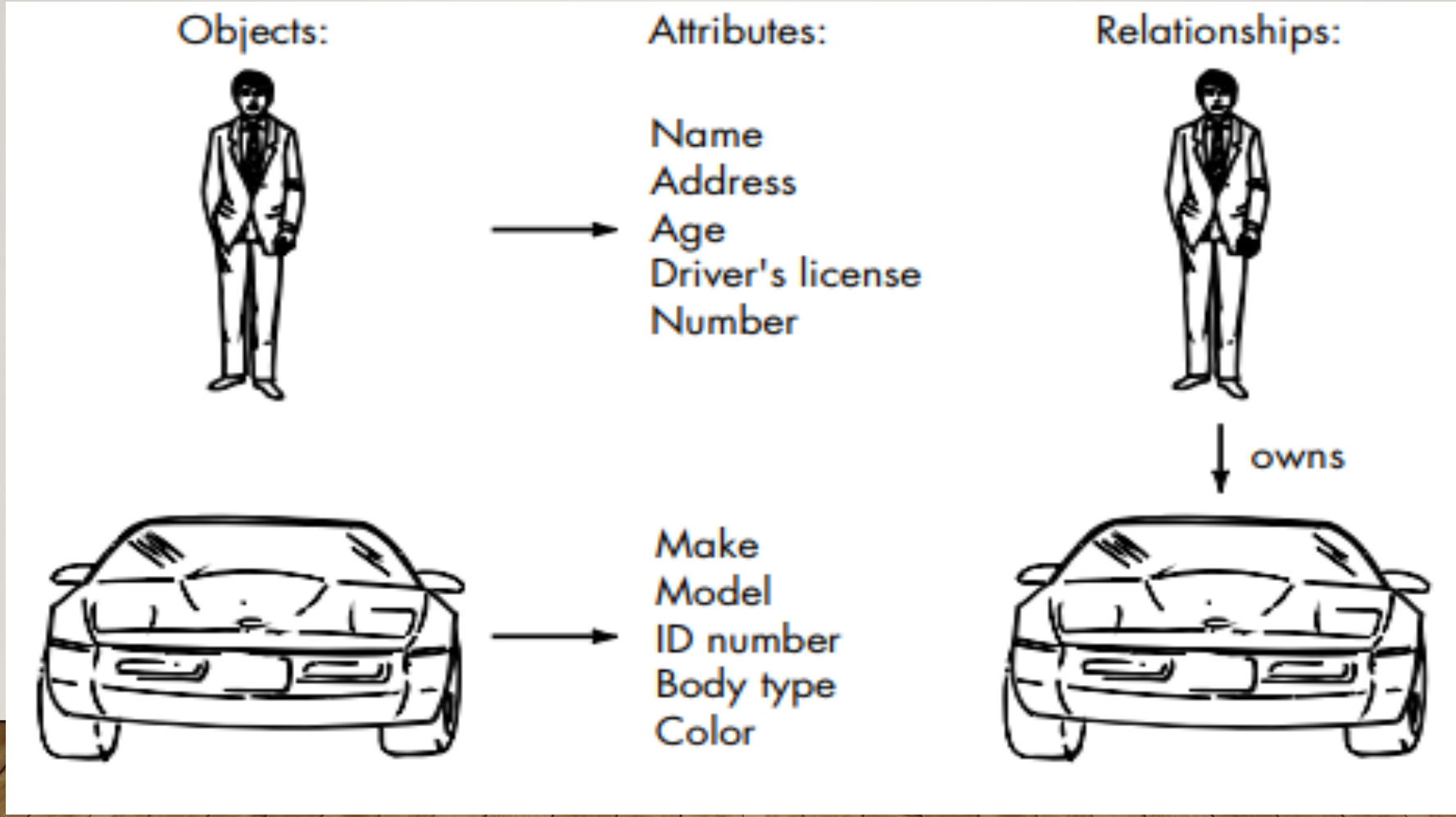
➢ -- It is not a flow chart

➢ -- Suppress logical decisions

ENTITY- RELATIONSHIP DIAGRAM (ER-DIAGRAM)

- The entity relation diagram (ERD) **depicts relationships between data objects**.
- The ERD is the notation that is used to **conduct the data modeling activity**.
- **The attributes of each data object noted in the ERD can be described using a data object description.**
- The entity relationship diagram **focuses solely on data** (and therefore satisfies the first operational analysis principles), **representing a "data network" that exists for a given system.**
- Unlike the **data flow diagram (which is used to represent how data are transformed), data modeling considers data independent of the processing that transforms the data.**

DATA OBJECTS, ATTRIBUTES & RELATIONSHIPS:



Contd.

- A data object is a representation of **almost any composite information that must be understood by software. By composite information, we mean something that has a number of different properties or attributes.**
- Therefore, **width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object**.
- A data object can be an **external entity** (e.g., anything that produces or consumes information), **a thing** (e.g., a report or a display), **an occurrence** (e.g., a telephone call) or **event** (e.g., an alarm), **a role** (e.g., salesperson), **an organizational unit** (e.g., accounting department), **a place** (e.g., a warehouse), or **a structure** (e.g., a file).

Contd.

- Attributes define **the properties of a data object** and take on one of three different characteristics.
- They can be used to:

➢ **--Name an instance of the data object,**

➢ **--Describe the instance, or**

➢ **--Make reference to another instance in another table**.

- In addition, one or more of the **attributes must be defined as an identifier**—that is, the identifier attribute becomes a **"key"** when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. **Referring to the data object car, a reasonable identifier might be the ID number.**

Contd.

- Data objects are connected to one another in different ways.
- **Consider two data objects, book and bookstore.**
- A connection is established between book and bookstore because the two objects are related.
- But what are the relationships?
- For example,

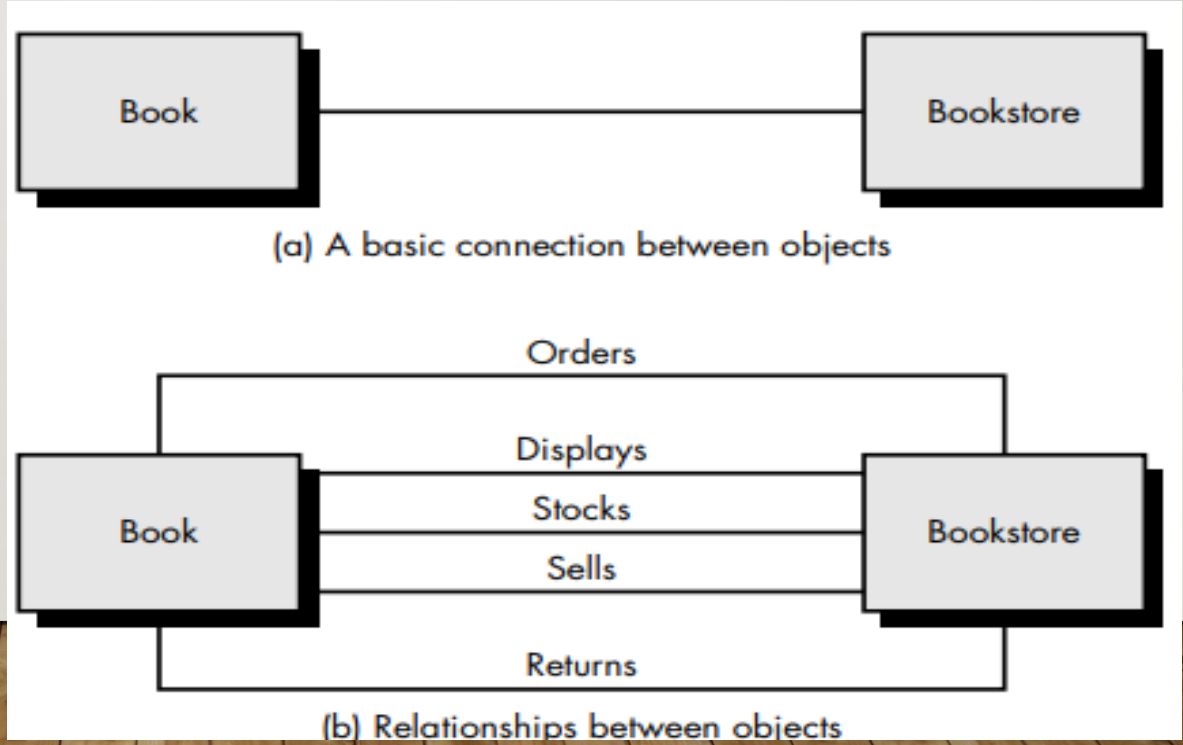
➢ A bookstore orders books.

➢ A bookstore displays books.

➢ A bookstore stocks books.

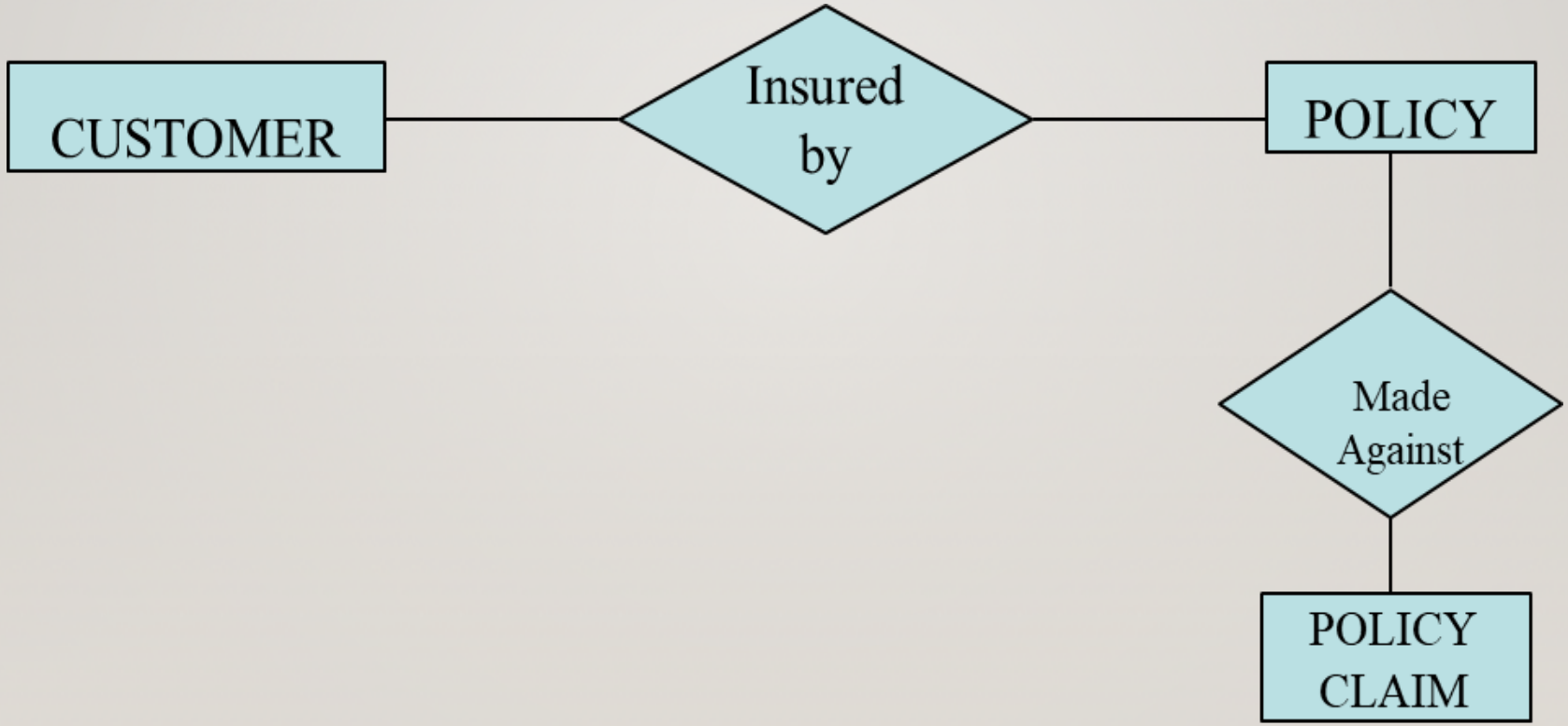
➢ A bookstore sells books.

➢ A bookstore returns books.

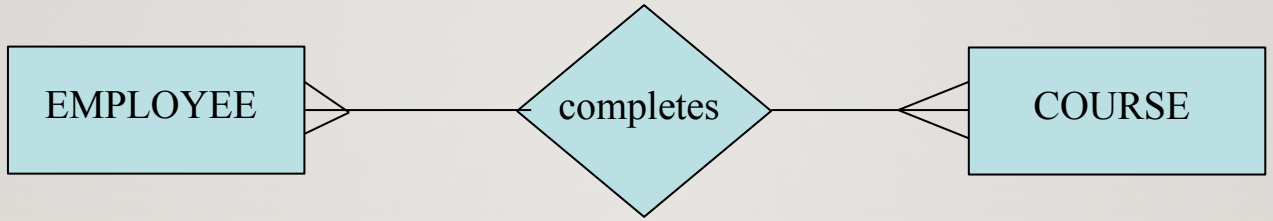


Relationships

- A relationship is a reason for associating two entity types. Binary relationships involve two entity types.
- A CUSTOMER is insured by a POLICY. A POLICY CLAIM is made against a POLICY.
- Relationships are represented by diamond notation in a ER diagram.



A training department is interested in tracking which training courses each of its employee has completed.

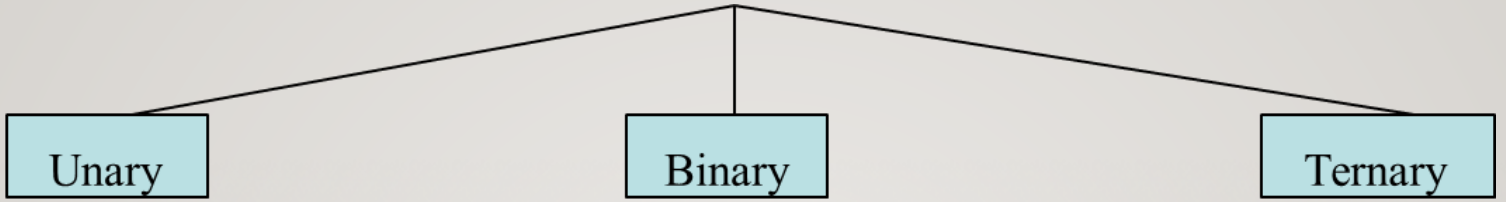


Many-to Many relationship

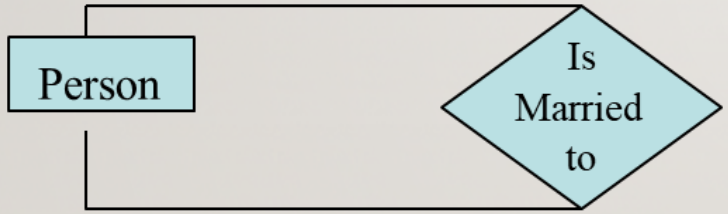
Each employee may complete more than one course,and each course may be completed by more than one employee.

Degree of relationship

It is the number of entity types that participates in that relationship.



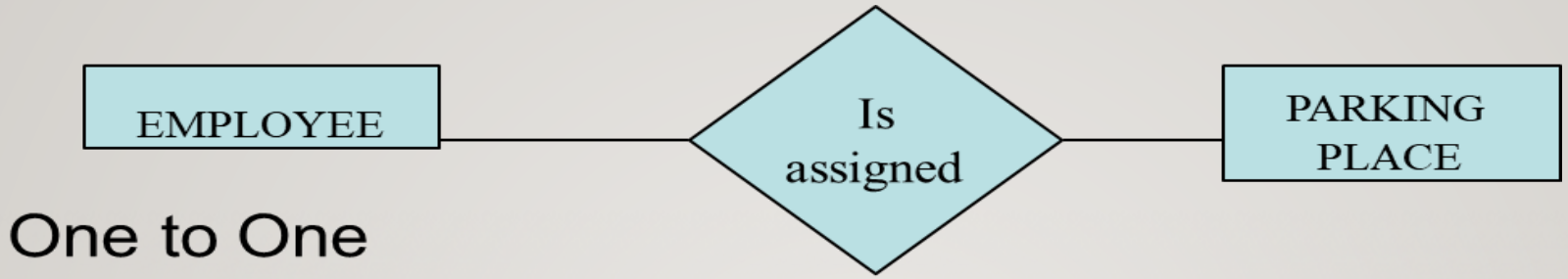
Unary Relationship



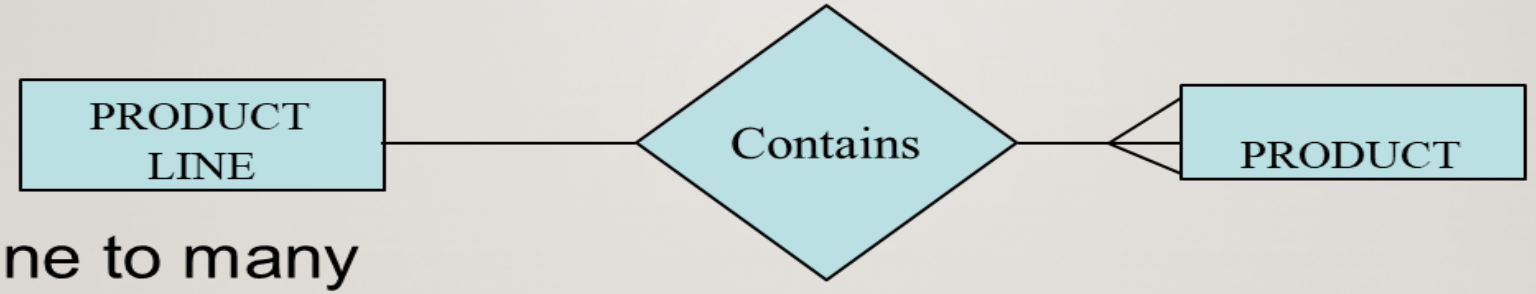
One to One

One to many

Binary Relationship



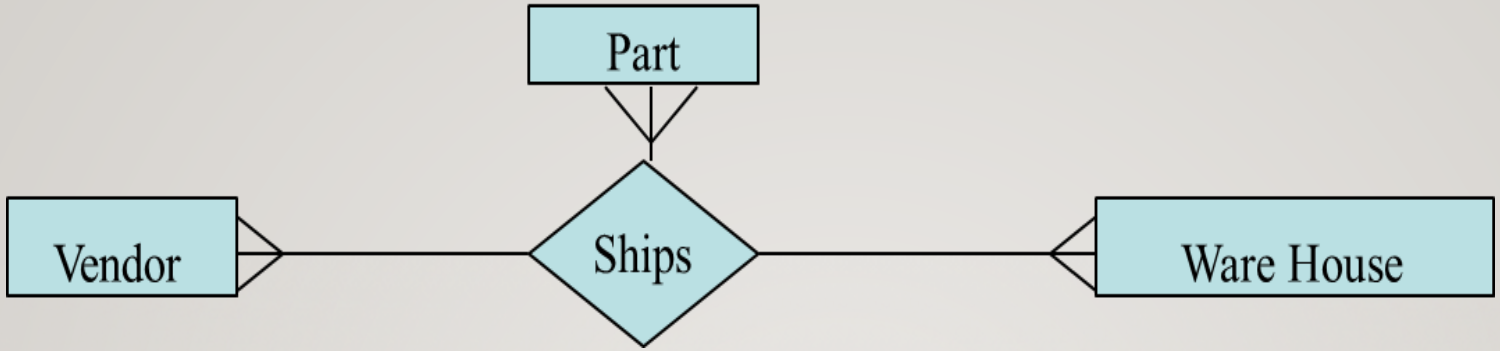
One to One



One to many

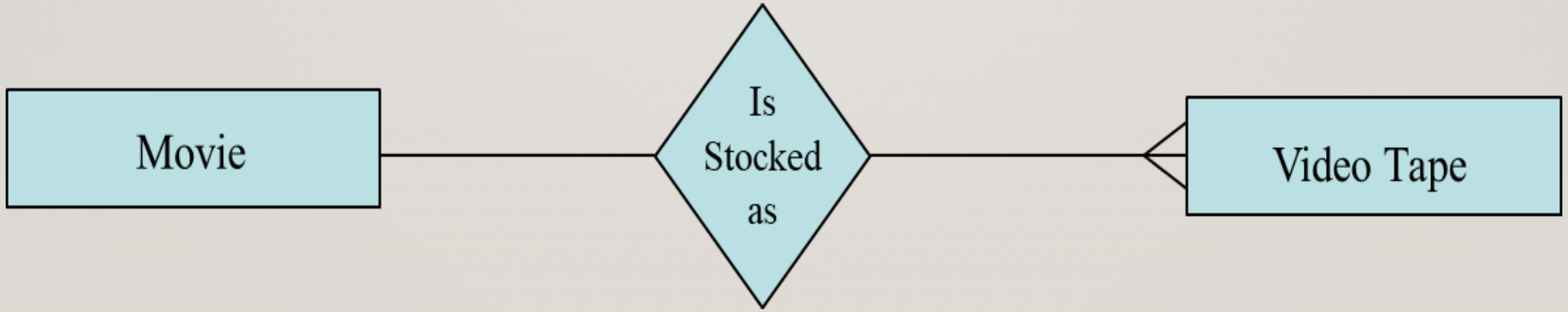
Many to many

Ternary relationship



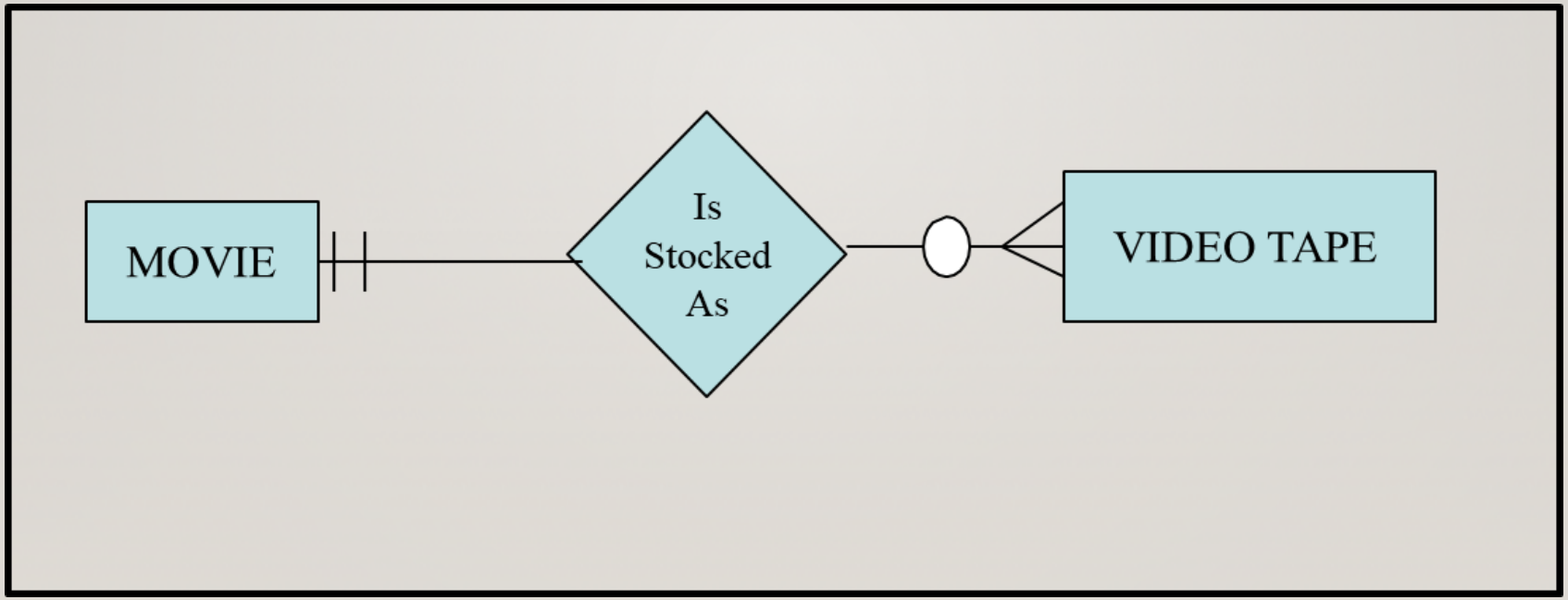
Cardinalities and optionality

Two entity types A & B connected by a relationship.
The cardinality of a relationship is the number of instances of entity B that can be associated with each instance of entity A



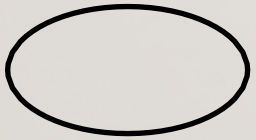
- **Minimum cardinality is the minimum number of instances of entity B that may be associated with each instance of entity A.**

- Minimum no. of tapes available for a movie is zero. We say VIDEO TAPE is an optional participant in the is-stocked-as relationship.



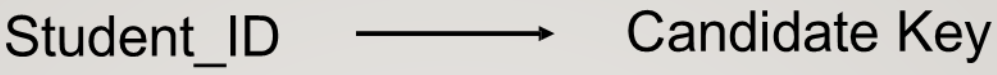
Attributes

- Each entity type has a set of attributes associated with it.
- An attribute is a property or characteristic of an entity that is of interest to organization.



Attribute

A candidate key is an attribute or combination of attributes that uniquely identifies each instance of an entity type.



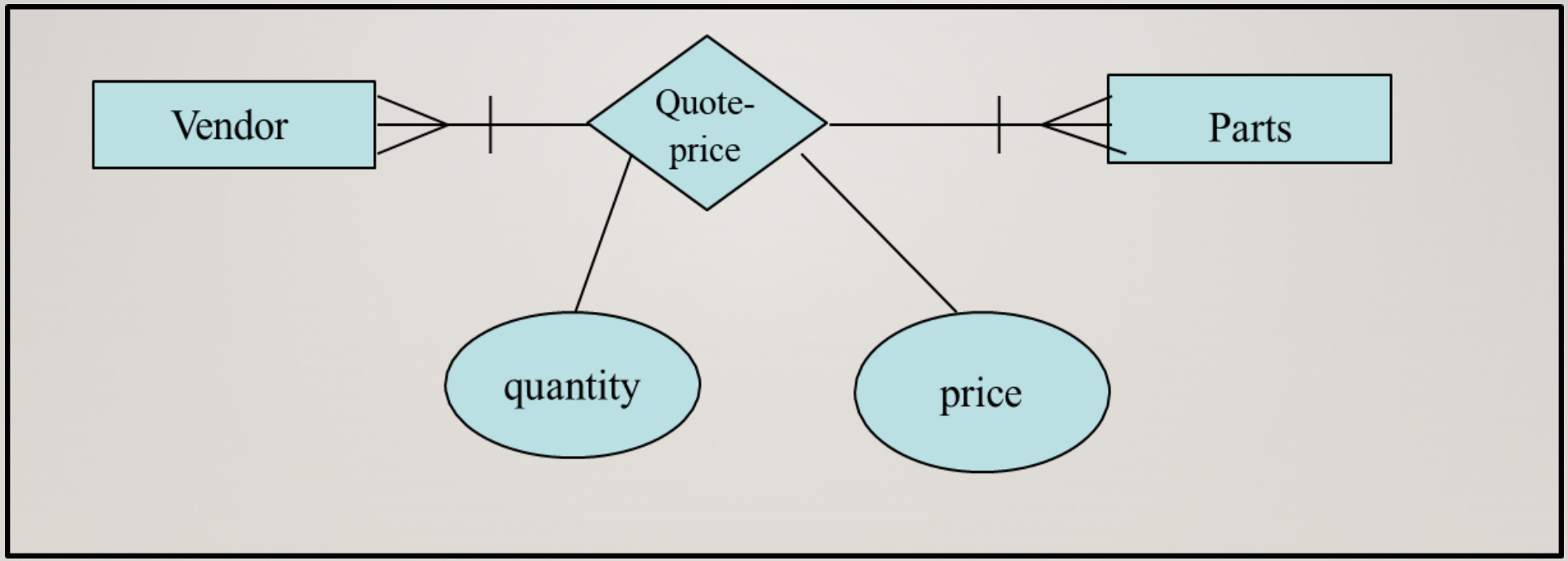
If there are more candidate keys, one of the key may be chosen as the Identifier.
It is used as unique characteristic for an entity type.

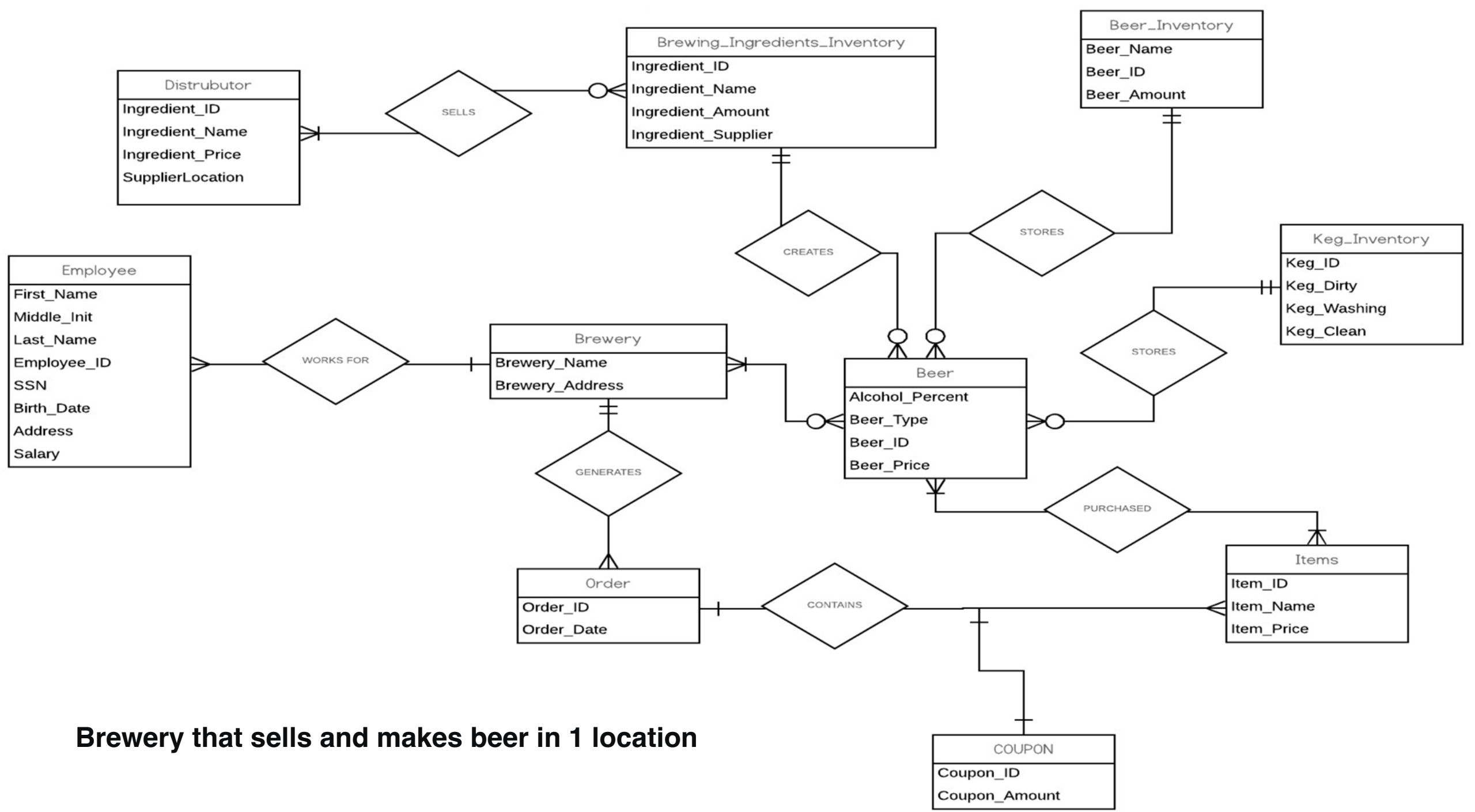
- One you have identified all the candidate keys, choose a **primary key**.
- Each strong entity in an E-R diagram has a primary key.
- You may have several candidate keys to choose from.

In addition, make sure that **the primary key has the following properties**:

1. A non-null value for each instance of the entity.
2. A unique value for each instance of an entity.
3. A non-changing value for the life of each entity instance.

Vendors quote prices for several parts along with quantity of parts. Draw an E-R diagram.





Brewery that sells and makes beer in 1 location

REQUIREMENT DOCUMENTATION

- This is the way of representing requirements in a consistent format.
- It is called **Software Requirement Specification(SRS)**.
- SRS serves many purpose depending upon who is writing it.

➢ --written by customer

➢ --written by developer

- **Serves as contract between customer & developer.**

NATURE OF SRS

- **The basic issues that the SRS writer(s) shall address are the following:**
  - --Functionality
  - --External Interfaces
  - --Performance
  - --Attributes
  - --Design Constraints imposed on an implementation.

- **SRS Should**
  - -- Correctly define all requirements
  - -- not describe any design details
  - -- not impose any additional constraints

CHARACTERISTICS OF SRS

- SRS should be:

1. Correct
2. Unambiguous
3. Complete
4. Consistent
5. Ranked for importance and/or stability
6. Verifiable
7. Modifiable
8. Traceable

Contd.

- **Correct:** An SRS is correct if and only if every requirement stated therein is one that the software shall meet.
- **Unambiguous:** An SRS is unambiguous if and only if, every requirement stated therein has only one interpretation.
- **Complete:** An SRS is complete if and only if, it includes the following elements-

(i) All significant requirements, whether related to functionality, performance, design constraints, attributes or external interfaces.

(ii) Responses to both valid & invalid inputs.

(iii) Full Label and references to all figures, tables and diagrams in the SRS and definition of all terms and units of measure.

Contd.

- **Consistent:** An SRS is consistent if and only if, no subset of individual requirements described in it conflict.
- **Ranked for importance and/or Stability:** If an identifier is attached to every requirement to indicate either the importance or stability of that particular requirement.
- **Verifiable:** An SRS is verifiable, if and only if, every requirement stated therein is verifiable.
- **Modifiable:** An SRS is modifiable, if and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining structure and style.
- **Traceable:** An SRS is traceable, if the origin of each of the requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation.

ADVANTAGES OF SRS

- Software SRS establishes the basic for agreement between the client and the supplier on what the software product will do.

➢A SRS provides a reference for validation of the final product.

➢A high-quality SRS is a prerequisite to high-quality software.

➢A high-quality SRS reduces the development cost.

PROBLEMS WITHOUT A SRS DOCUMENT

- The important problems that an organization would face if it does not develop an SRS document are as follows:

➢ **The system would not be implemented according to customer needs.**

➢ **Software developers would not know whether what they are developing is what exactly is required by the customer.**

➢ **It will be very difficult for the maintenance engineers to understand the functionality of the system**.

➢ **It will be very difficult for user document writers to write the users' manuals properly without understanding the SRS document.**

ORGANIZATION OF THE SRS

IEEE has published guidelines and standards to organize an SRS.

1. Introduction
    - 1.1 Purpose
    - 1.2 Scope
    - 1.3 Definition, Acronyms and abbreviations
    - 1.4 References
    - 1.5 Overview

2. The Overall Description

2.1 Product Perspective

- 2.1.1 System Interfaces
- 2.1.2 Interfaces
- 2.1.3 Hardware Interfaces
- 2.1.4 Software Interfaces
- 2.1.5 Communication Interfaces
- 2.1.6 Memory Constraints
- 2.1.7 Operations
- 2.1.8 Site Adaptation Requirements

2.2 Product Functions

2.3 User Characteristics

2.4 Constraints

2.5 Assumptions for dependencies

2.6 Apportioning of requirements

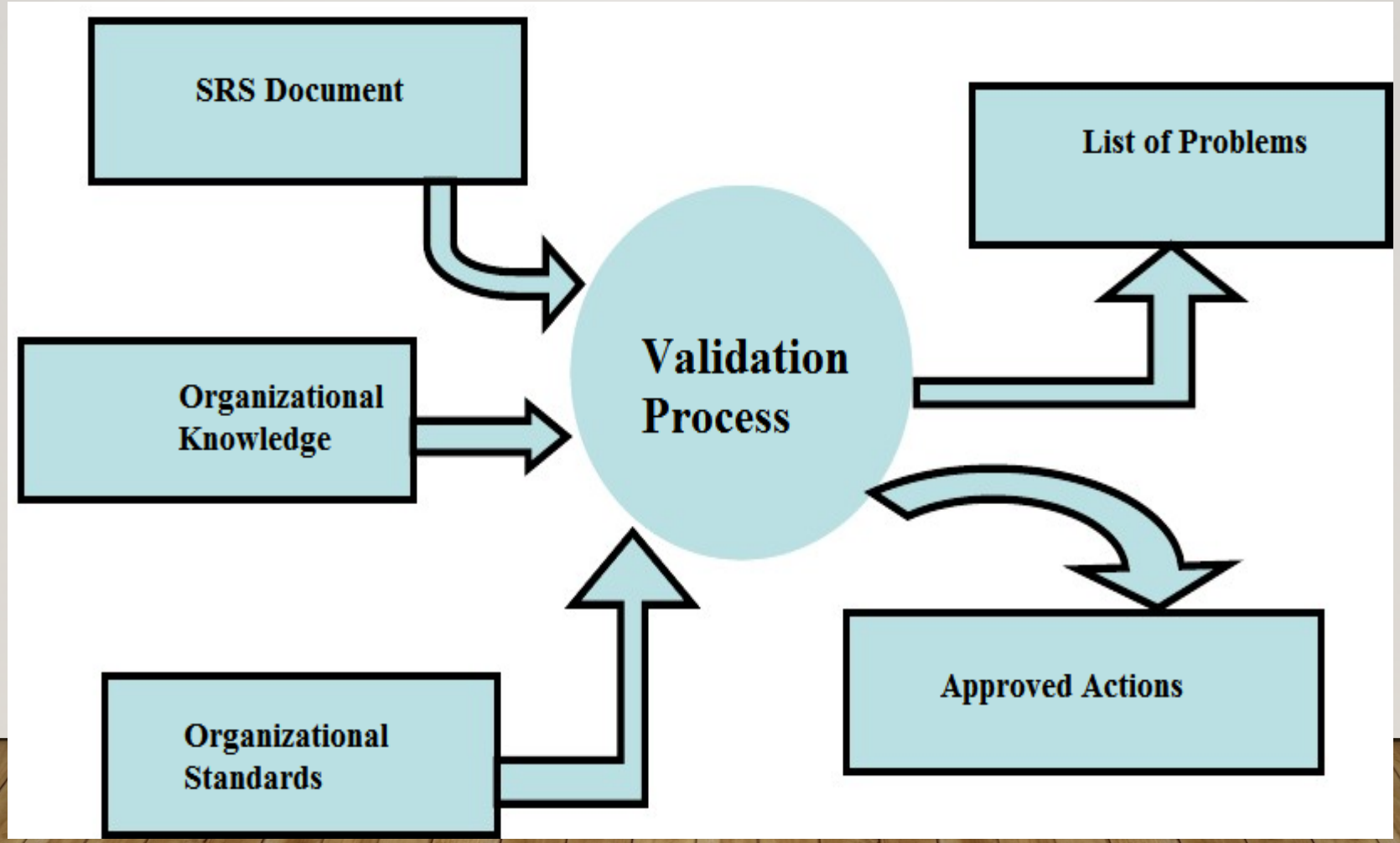
REQUIREMENT VALIDATION

After the completion of SRS document, we would like to check the document for:

- Completeness & consistency
- Conformance to standards
- Requirements conflicts
- Technical errors
- Ambiguous requirements

The objective of requirements validation is **to certify that the SRS document is an acceptable document of the system to be implemented.**

REQUIREMENT VALIDATION

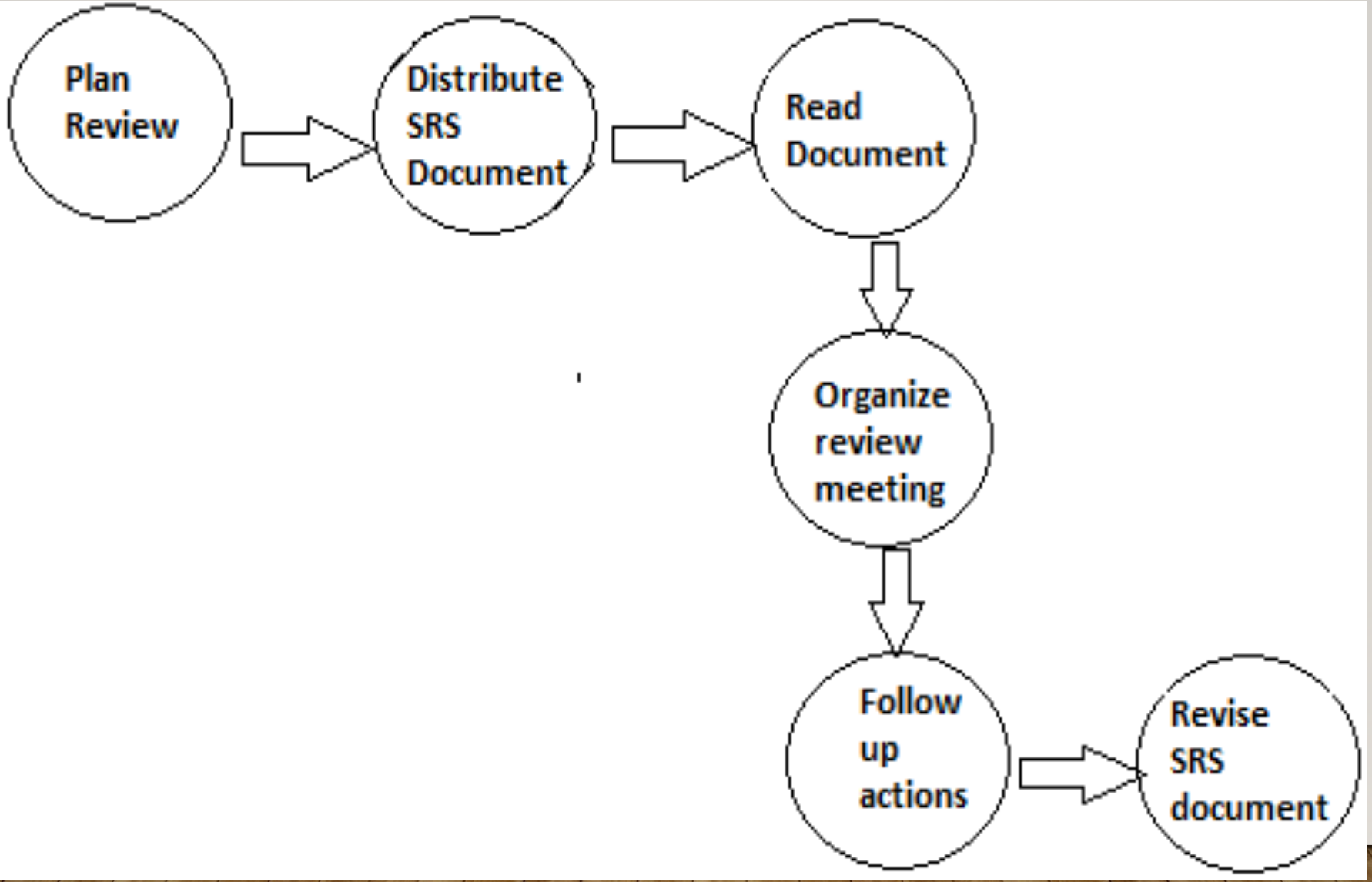


VALIDATION TECHNIQUES

- **There are two requirements validation techniques:-**

1. **Requirements Reviews**
2. **Prototyping**

REQUIREMENTS REVIEWS



PROBLEM ACTIONS IN REQUIREMENT VALIDATION

- Requirements clarification
- Missing information (find this information from stakeholders)
- Requirements conflicts (Stakeholders must negotiate to resolve this conflict)
- Unrealistic requirements
- Security issues

PROTOTYPING

- Validation prototype should be reasonably complete, efficient & should be used as the required system.

REVIEW CHECKLIST

- ✓ Understandability
- ✓ Redundancy
- ✓ Completeness
- ✓ Ambiguity
- ✓ Consistency
- ✓ Organization
- ✓ Conformance to standards
- ✓ Traceability

REQUIREMENT MANAGEMENT

- **Process of understanding and controlling changes to system requirements.**

- **ENDURING & VOLATILE REQUIREMENTS:**
- Enduring requirements: They are core requirements & are related to main activity of the organization. Example: issue/return of a book, cataloging etc.
- Volatile requirements: They are likely to change during software development life cycle or after delivery of the product
- **Requirement management planning is very critical and important for the success of any project.**

REQUIREMENT CHANGE MANAGEMENT

- Allocating adequate resources
- Analysis of requirements
- Documenting requirements
- Requirements traceability
- Establishing team communication
- Establishment of baseline

E-BOOKS

- **Software Engineering- A Practitioner's Approach** by **Roger S. Pressman**
- **Software Engineering** Ninth Edition by **Ian Sommerville**
- **Software Engineering** Third Edition by **K.K Aggarwal & Yogesh Singh**

SOFTWARE PROJECT PLANNING

- After the finalization of SRS, we would like to estimate size, cost and development time of the project.
- Also, in many cases, customer may like to know the cost and development time even prior to finalization of the SRS

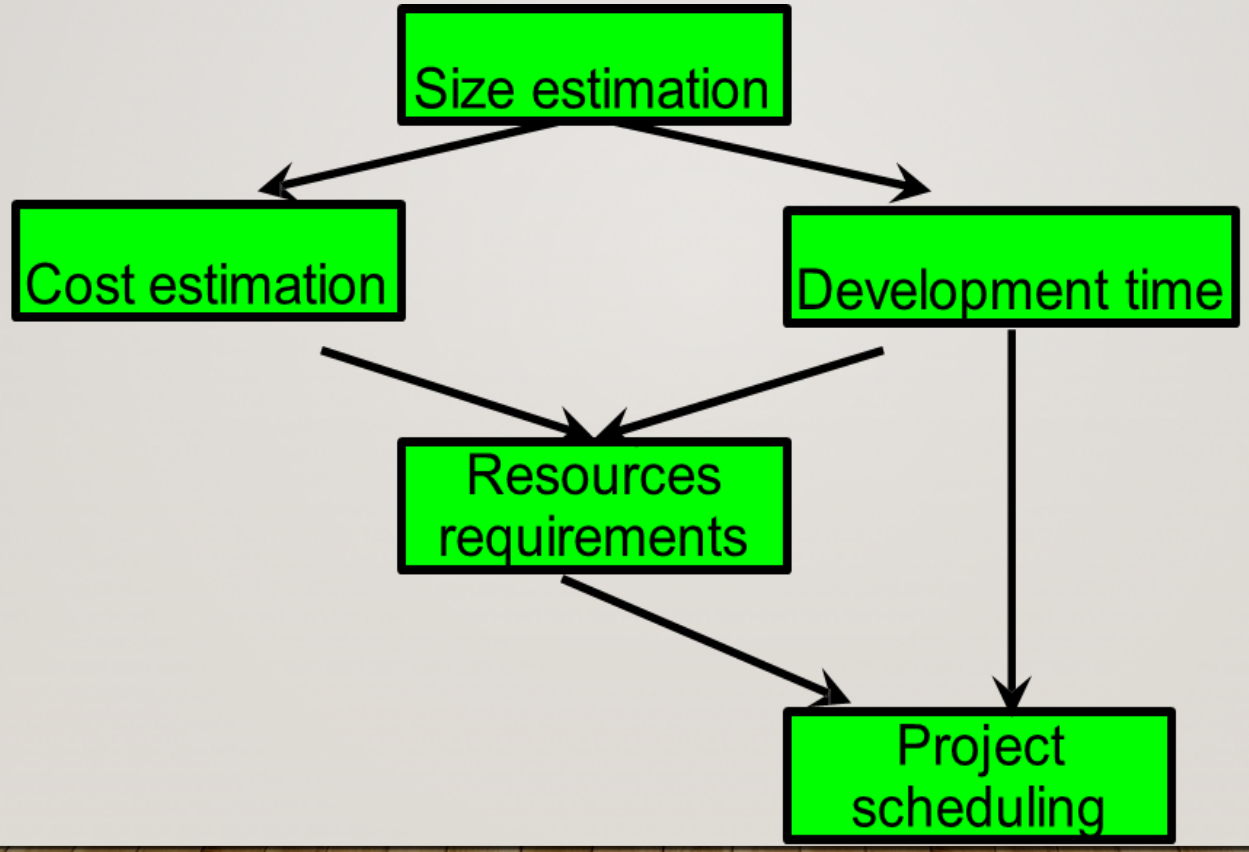
Contd.

In order to conduct a successful software project, we must understand:

- Scope of work to be done
- The risk to be incurred
- The resources required
- The task to be accomplished
- The cost to be expended
- The schedule to be followed

Activities during Software Project Planning

Software planning begins before technical work starts, continues as the software evolves from concept to reality, and culminates only when the software is retired.



SIZE ESTIMATION

- **Lines of Code (LOC)**

- If LOC is simply a count of the number of lines then figure shown below contains 18 LOC .

- When comments and blank lines are ignored, the program shown below contains **17 LOC.**

1.	int. sort (int x[ ], int n)
2.	{
3.	int i, j, save, im1;
4.	/*This function sorts array x in ascending order */
5.	If (n<2) return 1;
6.	for (i=2; i<=n; i++)
7.	{
8.	im1=i-1;
9.	for (j=1; j<=im; j++)
10.	if (x[i] < x[j])
11.	{
12.	Save = x[i];
13.	x[i] = x[j];
14.	x[j] = save;
15.	}
16.	}
17.	return 0;
18.	}

Function for sorting an array

Contd.

- Furthermore, if the main interest is the size of the program for specific functionality, it may be **reasonable to include executable statements**.
- The only executable statements in figure shown above were in lines 5-17 leading to a count of 13.
- The differences in the counts are 18 to 17 to 13.
- One can easily see the potential for major discrepancies for large programs with many comments or programs written in language that allow a large number of descriptive but non-executable statement.
- **Lines of code as:** *"A line of code is **any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line.** This specifically includes all lines containing program header, declaration, and executable and non-executable statements".*
- **This is the predominant definition for lines of code used by researchers. By this definition, figure shown above has 17 LOC.**

FUNCTION COUNT

Alan Albrecht while working for IBM, recognized the problem in size measurement in the 1970s, and developed a technique (which he called Function Point Analysis), which appeared to be a solution to the size measurement problem.

The principle of Albrecht's function point analysis (FPA) is that a system is decomposed into functional units.

▪ Inputs	:	information entering the system
▪ Outputs	:	information leaving the system
▪ Enquiries	:	requests for instant access to information
▪ Internal logical files	:	information held within the system
▪ External interface files	:	information held by other system that is used by the system being analyzed.

SPECIAL FEATURES

- Function point approach is **independent of the language, tools, or methodologies used for implementation**; i.e. they do not take into consideration programming languages, data base management systems, processing hardware or any other data base technology.
- Function points **can be estimated from requirement specification or design specification,** thus making it possible to estimate development efforts in early phases of development.
- Function points **are directly linked to the statement of requirements**; any change of requirements can easily be followed by a re-estimate.
- Function points are **based on the system user's external view of the system, non-technical users of the software system have a better understanding of what function points are measuring.**

COUNTING FUNCTION POINTS

Functional Units	Weighting factors		
	Low	Average	High
External Inputs (EI)	3	4	6
External Output (EO)	4	5	7
External Inquiries (EQ)	3	4	6
External logical files (ILF)	7	10	15
External Interface files (EIF)	5	7	10

Table 1:Functional units with weighting factors

UNADJUSTED FUNCTION POINT(UFP)

The weighting factors are identified for all functional units and multiplied with the functional units accordingly. The procedure for the calculation of Unadjusted Function Point (UFP) is given in table shown above.

Table 2: UFP calculation table

Functional Units	Count Complexity			Complexity Totals	Functional Unit Totals
External Inputs (EIs)		Low x 3	=		
		Average x 4	=		
		High x 6	=		
External Outputs (EOs)		Low x 4	=		
		Average x 5	=		
		High x 7	=		
External Inquiries (EQs)		Low x 3	=		
		Average x 4	=		
		High x 6	=		
External logical Files (ILFs)		Low x 7	=		
		Average x 10	=		
		High x 15	=		
External Interface Files (EIFs)		Low x 5	=		
		Average x 7	=		
		High x 10	=		
Total Unadjusted Function Point Count					

The procedure for the calculation of UFP in mathematical form is given below:

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

Where i indicate the row and j indicates the column of Table 1

W_{ij} : It is the entry of the i^{th} row and j^{th} column of the table 1

Zij : It is the count of the number of functional units of Type *i* that have been classified as having the complexity corresponding to column *j*.

Organizations that use function point methods develop a criterion for determining whether a particular entry is Low, Average or High.

Nonetheless, the determination of complexity is somewhat subjective.

$$\mathbf{FP = UFP * CAF}$$

Where **CAF** is **complexity adjustment factor** and is **equal to [0.65 +0.01 x ΣF_i].**
The **F_i (i=1 to 14)** are **the degree of influence** and are based on responses to questions noted shown in table 3 below.

Table 3 : Computing function points.

Rate each factor on a scale of 0 to 5.

Number of factors considered (F_i)

1. Does the system require reliable backup and recovery ?
2. Is data communication required ?
3. Are there distributed processing functions ?
4. Is performance critical ?
5. Will the system run in an existing heavily utilized operational environment ?
6. Does the system require on line data entry ?
7. Does the on line data entry require the input transaction to be built over multiple screens or operations ?
8. Are the master files updated on line ?
9. Is the inputs, outputs, files, or inquiries complex ?
10. Is the internal processing complex ?
11. Is the code designed to be reusable ?
12. Are conversion and installation included in the design ?
13. Is the system designed for multiple installations in different organizations ?
14. Is the application designed to facilitate change and ease of use by the user ?

FUNCTIONS POINTS MAY COMPUTE THE FOLLOWING IMPORTANT METRICS:

Productivity	=	FP / persons-months
Quality	=	Defects / FP
Cost	=	Rupees / FP
Documentation	=	Pages of documentation per FP

- These metrics are controversial and are not universally acceptable.
- There are standards issued by the International Functions Point User Group (IFPUG, covering the Albrecht method) and the United Kingdom Function Point User Group (UFPGU, covering the MK11 method).
- An ISO standard for function point method is also being developed.

EXAMPLE:1

Given the following values, compute function point when all complexity adjustment factor (CAF) and weighting factors are average.

User Input = 50
User Output = 40
User Inquiries = 35
User Files = 6
External Interface = 4

SOLUTION

➢ **Step-1: Calculate the degree of influence(F)**

F = 14 * scale

Scale varies from 0 to 5 according to character of Complexity Adjustment Factor (CAF). Below table shows scale:

0 - No Influence
1 - Incidental
2 - Moderate
3 - Average
4 - Significant
5 – Essential

As complexity adjustment factor is average (given in question), hence,

scale = 3.

Therefore, F = 14 * 3 **= 42**

➢ **Step -2: Calculate Complexity Adjustment Factor (CAF),**

CAF = 0.65 + (0.01 * F)

CAF = 0.65 + (0.01 * 42) **= 1.07**

➢ **Step-3: As weighting factors are also average (given in question) hence we will multiply each individual function point to corresponding values in TABLE.**

UFP = (50*4) + (40*5) + (35*4) + (6*10) + (4*7) **= 628**

Function Units	Low	Avg	High
EI	3	4	6
EO	4	5	7
EQ	3	4	6
ILF	7	10	15
EIF	5	7	10

➢ **Step -4: Calculate Function Point**.

FP = UFP * CAF

Therefore, Function Point = 628 * 1.07 **= 671.96**

EXAMPLE:2

An application has the following:

10 low external inputs, 12 high external outputs, 20 low internal logical files, 15 high external interface files, 12 average externalinquiries, and a value of complexity adjustment factor of 1.10.

What are the **unadjusted and adjusted function point counts(FP)**?

SOLUTION

According to question, there is 10 low EI, 12 high EO, 20 low ILF, 15 high EIF and 12 average EQ

Unadjusted function point counts may be calculated using as:

$= 10 \times 3 + 12 \times 7 + 20 \times 7 + 15 \times 10 + 12 \times 4$

$= 30 + 84 + 140 + 150 + 48$

$= \mathbf{452}$

FP $= UFP \times CAF$

$= 452 \times 1.10 = \mathbf{497.2}$

EXAMPLE: 3

Consider a project with these parameters:

(i) External Inputs:
- (a)10 with low complexity
- (b)15 with average complexity
- (c) 17 with high complexity

(ii) External Outputs:
- (a)6 with low complexity
- (b)13 with high complexity

(iii) External Inquiries:
- (a) 3 with low complexity
- (b) 4 with average complexity
- (c) 2 high complexity

(iv) Internal logical files:

(a)2 with average complexity

(b)1 with high complexity

(v) External Interface files:

(a) 9 with low complexity

In addition to above, system requires

i. Significant data communication

ii. Performance is very critical

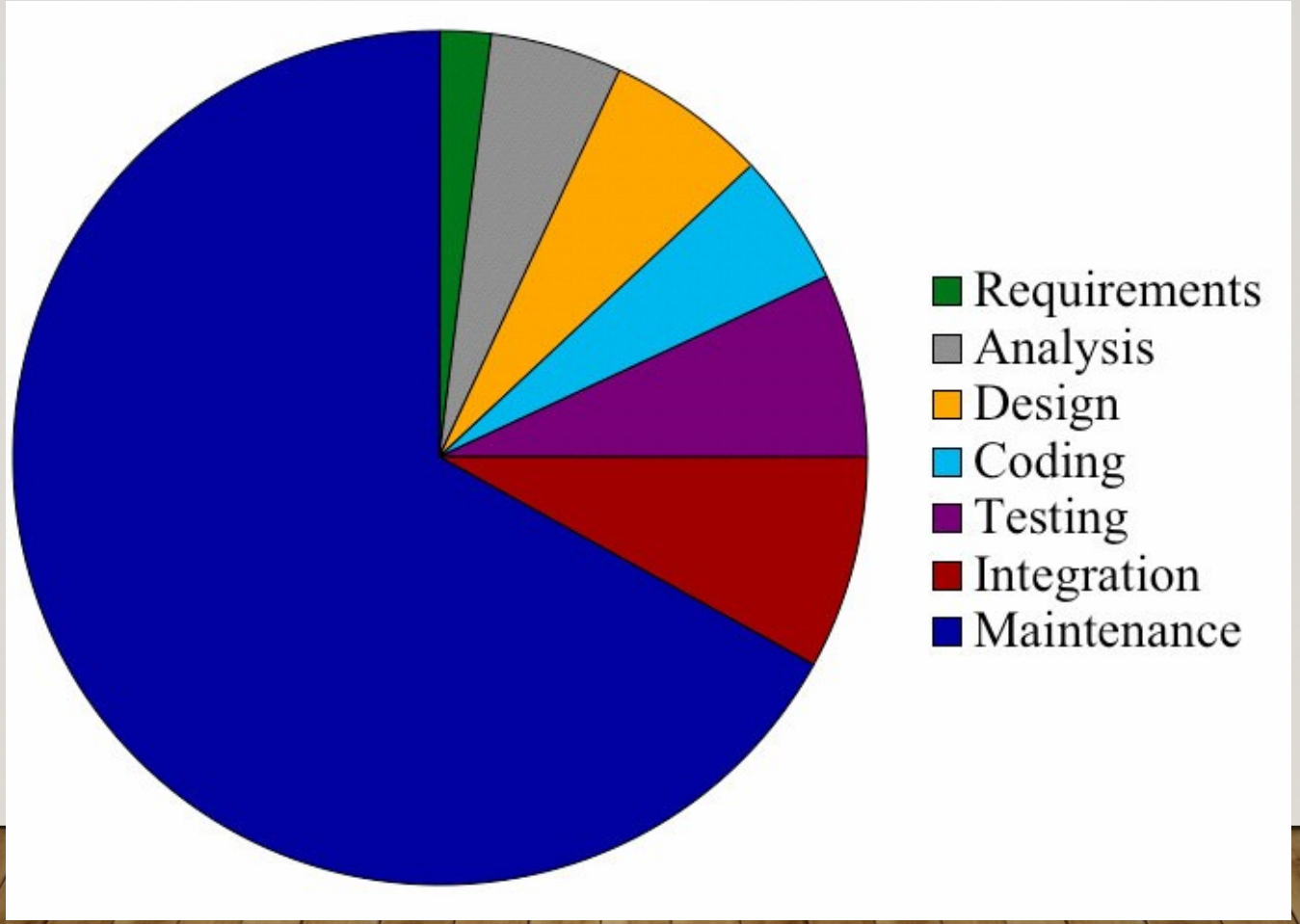
iii. Designed code may be moderately reusable

iv. System is not designed for multiple installation in different organizations.

Other complexity adjustment factors are treated as average.

Compute the function points for the project.

RELATIVE COST OF SOFTWARE PHASES



CONSTRUCTIVE COST MODEL(COCOMO MODEL)

- The COCOMO Model is a procedural cost estimate model for software projects and is often used as a process of reliably predicting the various parameters associated with making a project such as **size, effort, cost, time, and quality.** It was proposed by **Barry Boehm in 1981**.
- The key parameters that define the quality of any software products, which are also an outcome of the COCOMO are primarily Effort and schedule:

➢**--Effort:** Amount of labor that will be required to complete a task. It is measured in person-months units.

➢**--Development Time:** This simply means the amount of time required for the completion of the job, which is, of course, proportional to the effort put in. It is measured in the units of time such as weeks, and months.

CHARACTERISTICS OF DIFFERENT PROJECT TYPES

- Organic
- Semi-Detached
- Embedded

Mode	Project size	Nature of Project	Innovation	Deadline of the project	Development Environment
Organic	Typically 2-50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Little	Not tight	Familiar & In house
Semi detached	Typically 50-300 KLOC	Medium size project, Medium size team, Average previous experience on similar project. For example: Utility systems like compilers, database systems, editors etc.	Medium	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For example: ATMs, Air Traffic Control etc.	Significant	Tight	Complex Hardware/ customer Interfaces required

TYPES OF COCOMO MODEL

- Basic
- Intermediate
- Detailed

BASIC- COCOMO MODEL

Basic COCOMO model takes the form

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (E)^{d_b}$$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a_b, b_b, c_b and d_b are given in table below.

Software Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table: Basic COCOMO coefficient

Contd.

When effort and development time are known, the average staff size to complete the project may be calculated as:

$$\text{Average staff size } (SS) = \frac{E}{D} Persons$$

When project size is known, the productivity level may be calculated as:

$$\text{Productivity } (P) = \frac{KLOC}{E} KLOC / PM$$

EXAMPLE: 4.5

Suppose that a project was estimated to be **400 KLOC**. Calculate the effort and development time for each of the three modes i.e., organic, semidetached and embedded.

SOLUTION

The basic COCOMO equation take the form:

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (E)^{d_b}$$

Estimated size of the project = 400 KLOC

(i) Organic mode

$$E = 2.4(400)^{1.05} = 1295.31 \text{ PM}$$

$$D = 2.5(1295.31)^{0.38} = 38.07 \text{ M}$$

(ii) Semidetached mode

$$E = 3.0(400)^{1.12} = 2462.79 \text{ PM}$$

$$D = 2.5(2462.79)^{0.35} = 38.45 \text{ M}$$

(iii) Embedded mode

$$E = 3.6(400)^{1.20} = 4772.81 \text{ PM}$$

$$D = 2.5(4772.8)^{0.32} = 38 \text{ M}$$

EXAMPLE: 4.6

A project size of **200 KLOC** is to be developed.

Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the effort, development time, average staff size and productivity of the project.

The **semi-detached mode** is the most appropriate mode; keeping in view the size, schedule and experience of the development team.

Hence $E = 3.0(200)^{1.12} = 1133.12$ PM

$D = 2.5(1133.12)^{0.35} = 29.3$ M

Average staff size $(SS) = \frac{E}{D} Persons$

$= \frac{1133.12}{29.3} = 38.67 Persons$

$$\text{Productivity} = \frac{KLOC}{E} = \frac{200}{1133.12} = 0.1765\, KLOC / PM$$

$$P = 176\, LOC / PM$$

INTERMEDIATE MODEL

Cost drivers

(i) Product Attributes

- ➢ Required s/w reliability
- ➢ Size of application database
- ➢Complexity of the product

(*ii*) Hardware Attributes

- ➢ Run time performance constraints
- ➢ Memory constraints
- ➢ Virtual machine volatility
- ➢ Turnaround time

Contd.

(iii) Personal Attributes

- Analyst capability
- Programmer capability
- Application experience
- Virtual m/c experience
- Programming language experience

(iv) Project Attributes

- Modern programming practices
- Use of software tools
- Required development Schedule

Multipliers for Effort Adjustment Factor (EAF) of different cost drivers

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Product Attributes						
RELY	0.75	0.88	1.00	1.15	1.40	--
DATA	--	0.94	1.00	1.08	1.16	--
CPLX	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME	--	--	1.00	1.11	1.30	1.66
STOR	--	--	1.00	1.06	1.21	1.56
VIRT	--	0.87	1.00	1.15	1.30	--
TURN	--	0.87	1.00	1.07	1.15	--

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Personnel Attributes						
ACAP	1.46	1.19	1.00	0.86	0.71	--
AEXP	1.29	1.13	1.00	0.91	0.82	--
PCAP	1.42	1.17	1.00	0.86	0.70	--
VEXP	1.21	1.10	1.00	0.90	--	--
LEXP	1.14	1.07	1.00	0.95	--	--
Project Attributes						
MODP	1.24	1.10	1.00	0.91	0.82	--
TOOL	1.24	1.10	1.00	0.91	0.83	--
SCED	1.23	1.08	1.00	1.04	1.10	--

Intermediate COCOMO equations

$$E = a_i (KLOC)^{b_i} \text{ X } EAF$$

$$D = c_i (E)^{d_i}$$

Note: **EAF** is **Effort Adjustment Factor** calculated by multiplying the value of different cost drivers with there respective ratings(using table shown on previous two slides)

Project	i a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Table: Coefficients for intermediate COCOMO

EXAMPLE: 4.7

A new project with estimated **400 KLOC embedded system** has to be developed.

Project manager has a choice of hiring from two pools of developers:

Very highly capable with very little experience in the programming language being used.

Or

Developers of low quality but a lot of experience with the programming language.

What is the impact of hiring all developers from one or the other pool ?

SOLUTION

This is the case of embedded mode and model is intermediate COCOMO.

Hence $$E = a_i (KLOC)^{b_i} \text{ X } EAF$$

$$= 2.8\ (400)^{1.20} \text{ X EAF} = 3712 \text{ PM}$$

Case I: Developers are very highly capable with very little experience in the programming being used.

EAF = 0.82 x 1.14 = 0.9348

E = 3712 x .9348 = 3470 PM

D = $2.5\ (3470)^{0.32}$ = 33.9 M

Highly Capable means AEXP = 0.82

&

Very Little Experience means LEXP = 1.14

Case II: Developers are of low quality but lot of experience with the programming language being used.

EAF = 1.29 x 0.95 = 1.22

E = 3712 x 1.22 = 4528 PM

D = $2.5 (4528)^{0.32}$ = 36.9 M

Low Quality means AEXP = 1.29

&

Lot of Experience (high) means LEXP = 0.95

Case II requires more effort and time. **Hence, low quality developers with lot of programming language experience could not match with the performance of very highly capable developers with very little experience.**

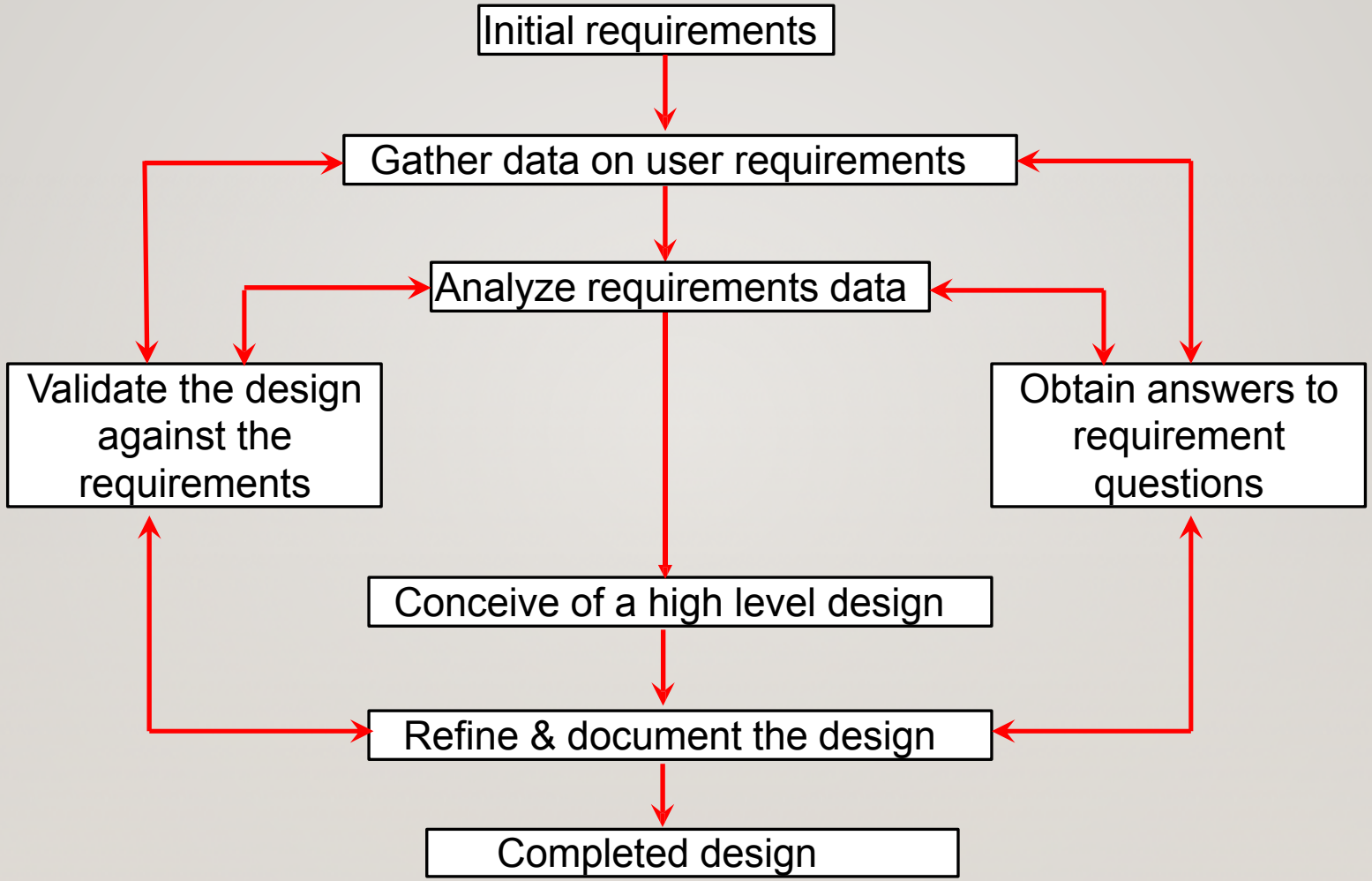
GLA UNIVERSITY
MATHURA
Recognised by UGC Under Section 2(f)
ऋते ज्ञानान्न मुक्तिः
Accredited with A+ Grade by NAAC
3.46 Score
12-B Status from UGC

SOFTWARE DESIGN

INTRODUCTION

- Software Design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.
- It produces a solution to a problem given in the SRS document.
- **The output of the design phase is a Software Design Document (SDD).**

Fig. 1 : Design framework

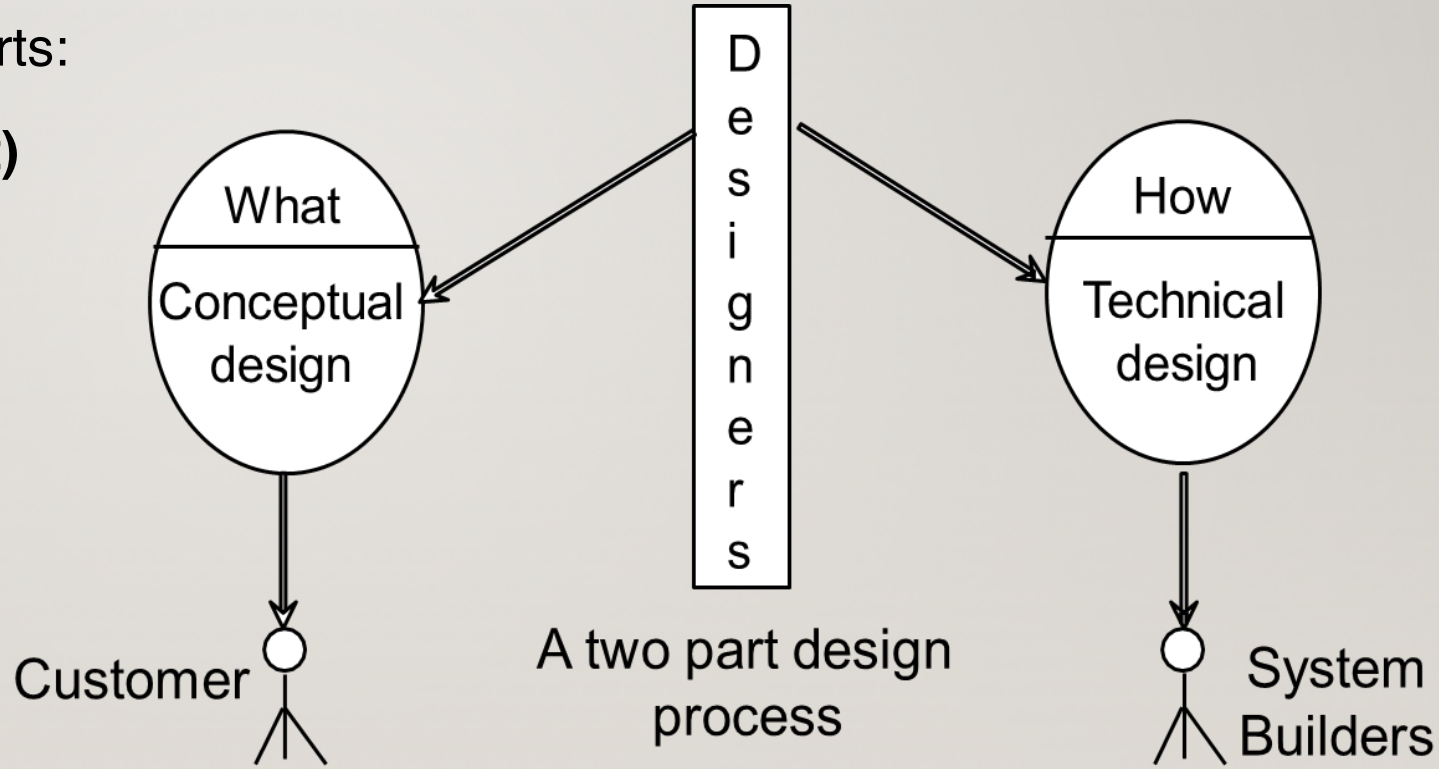


DESIGN SATISFY BOTH CUSTOMERS & DEVELOPERS

- Design process has two parts:

➢**Conceptual Design (What)**

➢**Technical Design (How)**



CONCEPTUAL DESIGN ANSWERS

- Where will the data come from ?
- What will happen to data in the system?
- How will the system look to users?
- What choices will be offered to users?
- What is the timings of events?
- How will the reports & screens look like?

TECHNICAL DESIGN DESCRIBES

- Hardware configuration
- Software needs
- Communication interfaces
- I/O of the system
- Software architecture
- Network architecture
- Any other thing that translates the requirements in to a solution to the customer's problem.

OBJECTIVES OF SYSTEM DESIGN

- **Correctness:** Software design should be correct as per requirement.
- **Completeness:** The design should have all components like data structures, modules, and external interfaces, etc.
- **Efficiency:** Resources should be used efficiently by the program.
- **Flexibility:** Able to modify on changing needs.
- **Consistency:** There should not be any inconsistency in the design.
- **Maintainability:** The design should be so simple so that it can be easily maintainable by other designers.

MODULARITY

- **Modularization is the process of dividing a software system into multiple independent modules where each module works independently.**
- There are many advantages of Modularization in software engineering. Some of these are given below:
  - ➢Easy to understand the system.
  - ➢System maintenance is easy.
  - ➢A module can be used many times as their requirements. No need to write it again and again.
- A modular system consist of well-defined manageable units with well defined interfaces among the units.

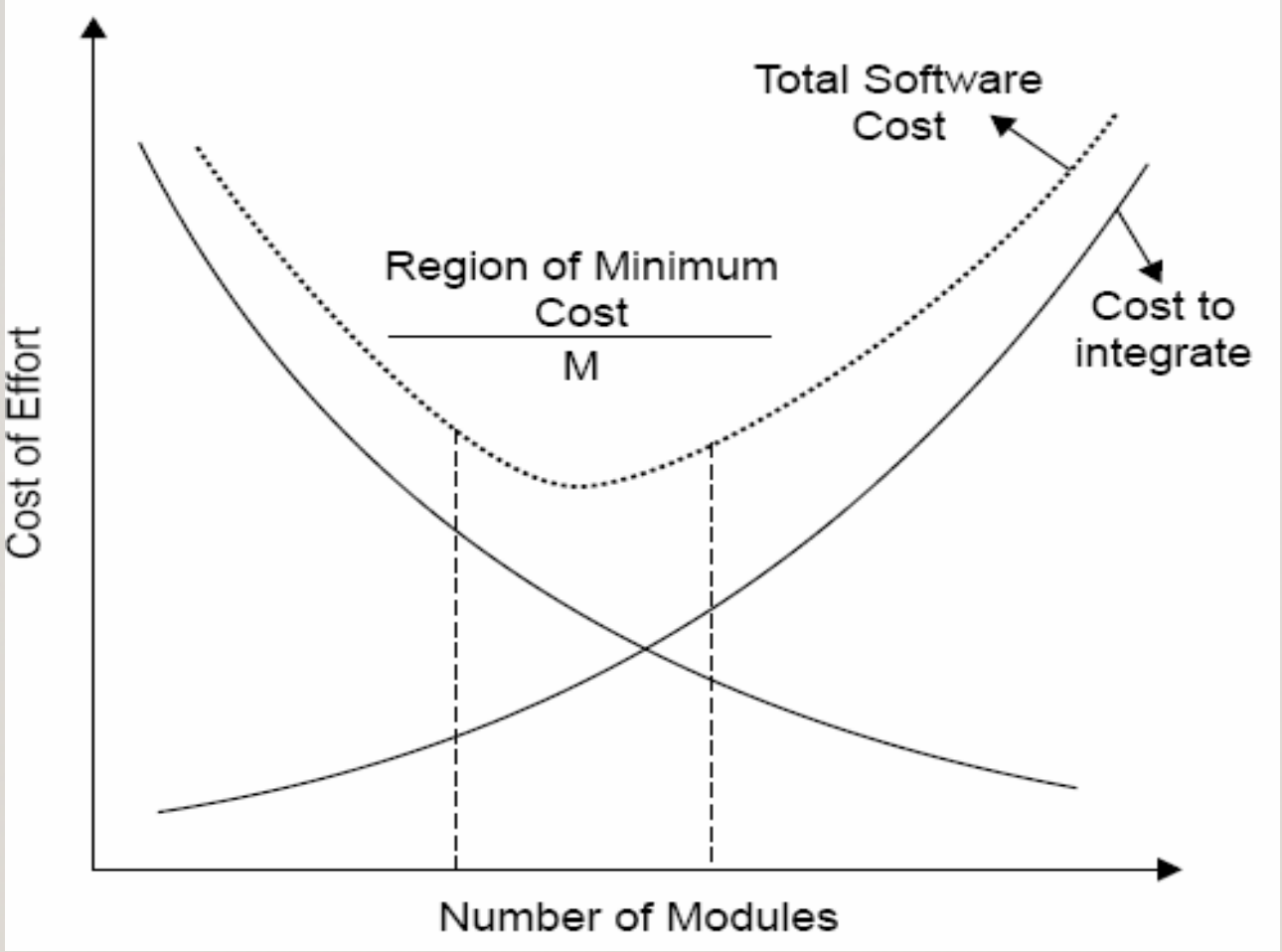
Contd.

- Modularity is the single attribute of software that allows a program to be intellectually manageable.
- It enhances design clarity, which in turn eases implementation, debugging, testing, documenting and maintenance of the software product.

PROPERTIES OF MODULARIZATION

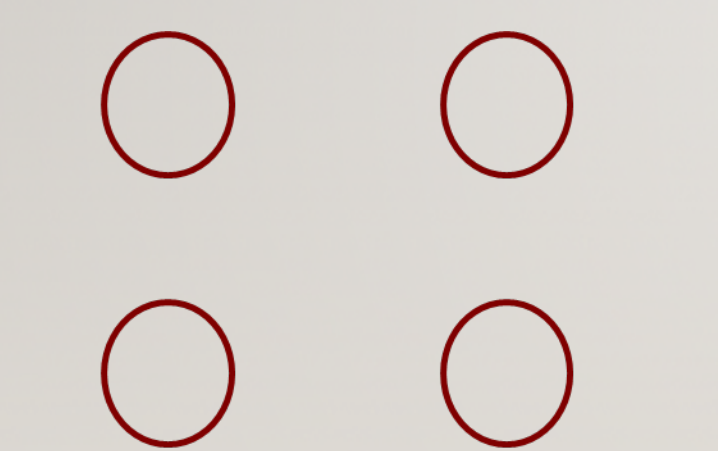
- Well defined subsystem
- Well defined purpose
- Can be separately compiled and stored in a library.
- Module can use other modules
- Module should be easier to use than to build
- Simpler from outside than from the inside.

Modularity and software cost



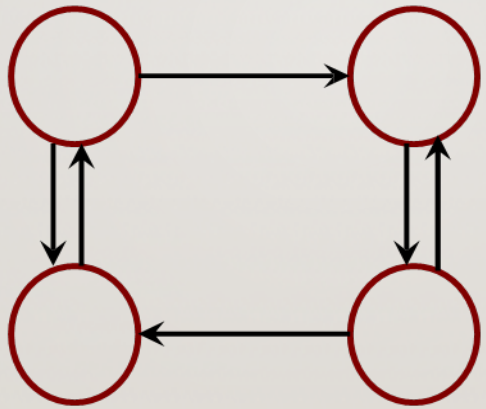
COUPLING

- Coupling is the measure of the degree of interdependence between the modules.
- **A good software will have low coupling.**



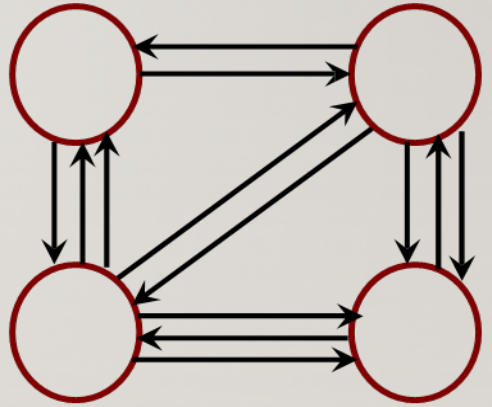
(Uncoupled : no dependencies)

(a)



Loosely coupled: some dependencies

(b)



Highly coupled: many dependencies

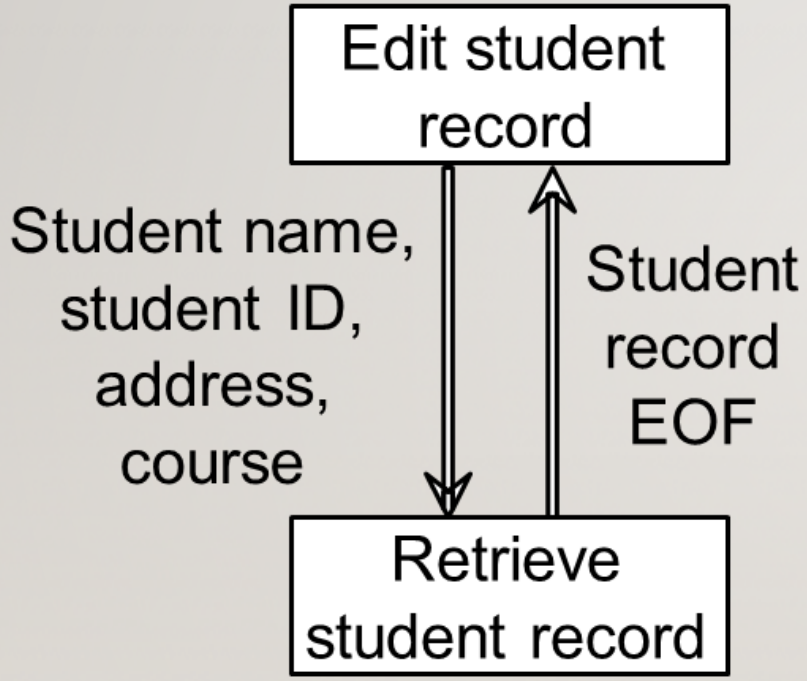
(c)

THIS CAN BE ACHIEVED BY

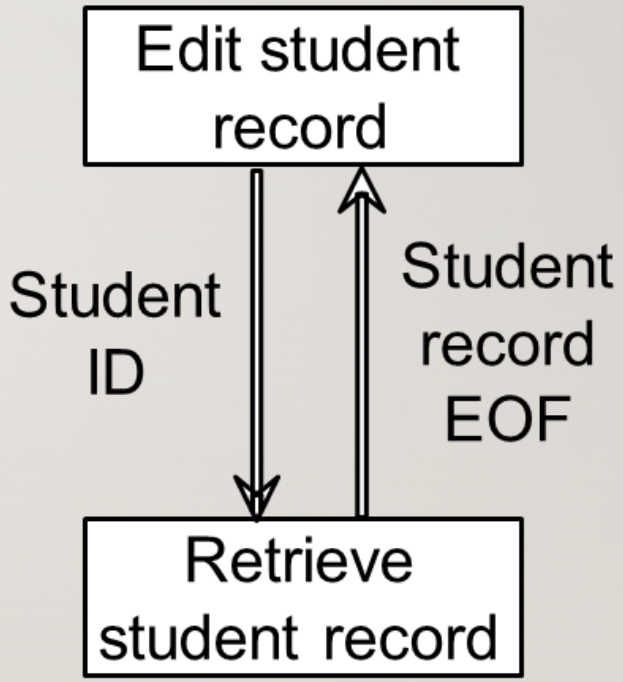
- Controlling the number of parameters passed amongst modules.
- Avoid passing undesired data to calling module.
- Maintain parent/child relationship between calling & called modules.
- Pass data, not the control information.

Example of coupling

Consider the example of editing a student record in a 'student information system'.



Poor design: Tight Coupling



Good design: Loose Coupling

TYPES OF COUPLING

Data coupling	Best
Stamp coupling	
Control coupling	
External coupling	
Common coupling	
Content coupling	Worst

DATA COUPLING

- If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled.
- In data coupling, the components are independent of each other and communicate through data.

STAMP COUPLING

- In stamp coupling, the complete data structure is passed from one module to another module.

CONTROL COUPLING

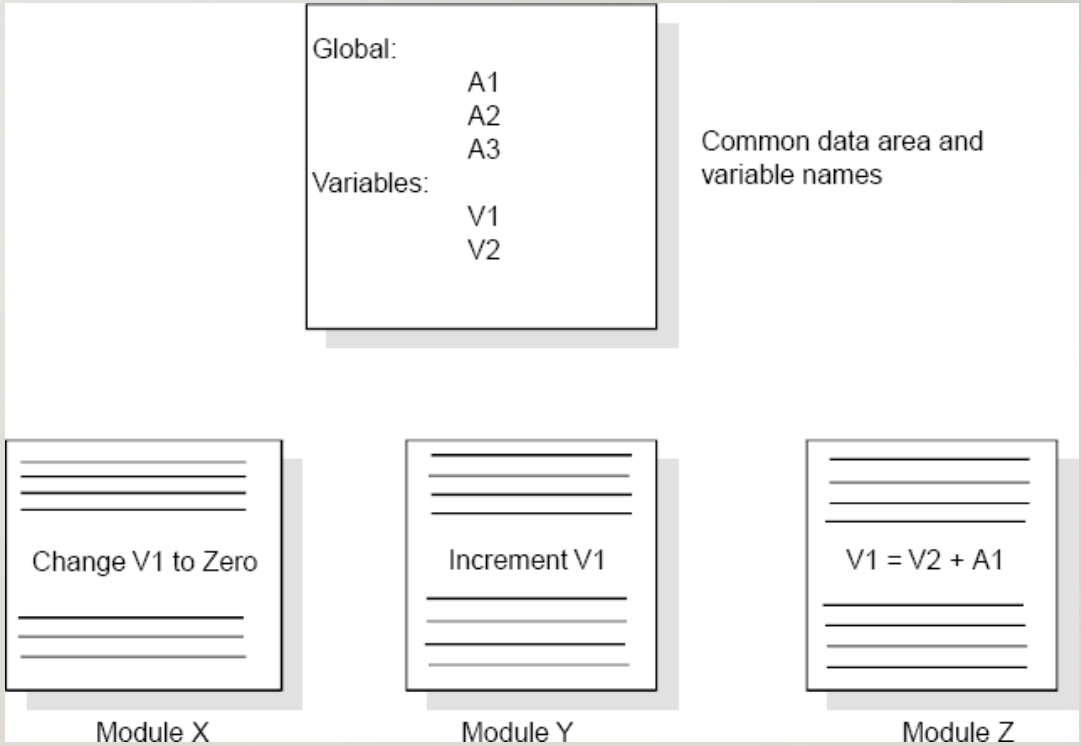
- If the modules communicate by passing control information, then they are said to be control coupled.
- It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality.

EXTERNAL COUPLING

- In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware.
- Ex- protocol, external file, device format, etc.

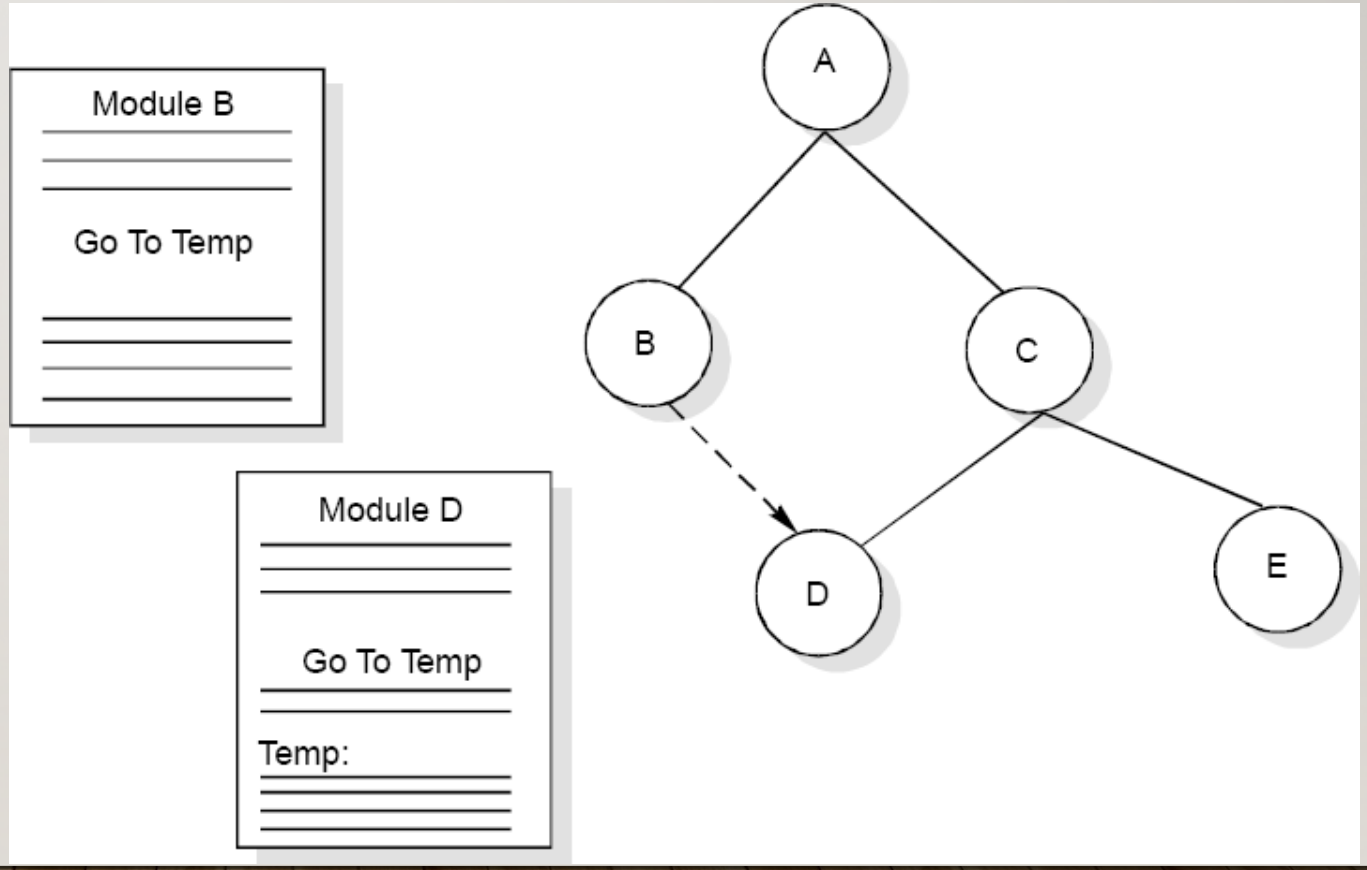
COMMON COUPLING

- **The modules have shared data such as global data structures.** The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change.
- So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses, and reduced maintainability.



CONTENT COUPLING

- In a content coupling, one module can modify the data of another module, or control flow is passed from one module to the other module.
- This is the worst form of coupling and should be avoided.



COHESION

- Cohesion is a measure of the **degree to which the elements of the module are functionally related.**
- It is the degree to which all elements directed towards performing a single task are contained in the component.
- Basically, cohesion is the internal glue that keeps the module together.
- **A good software design will have high cohesion.**

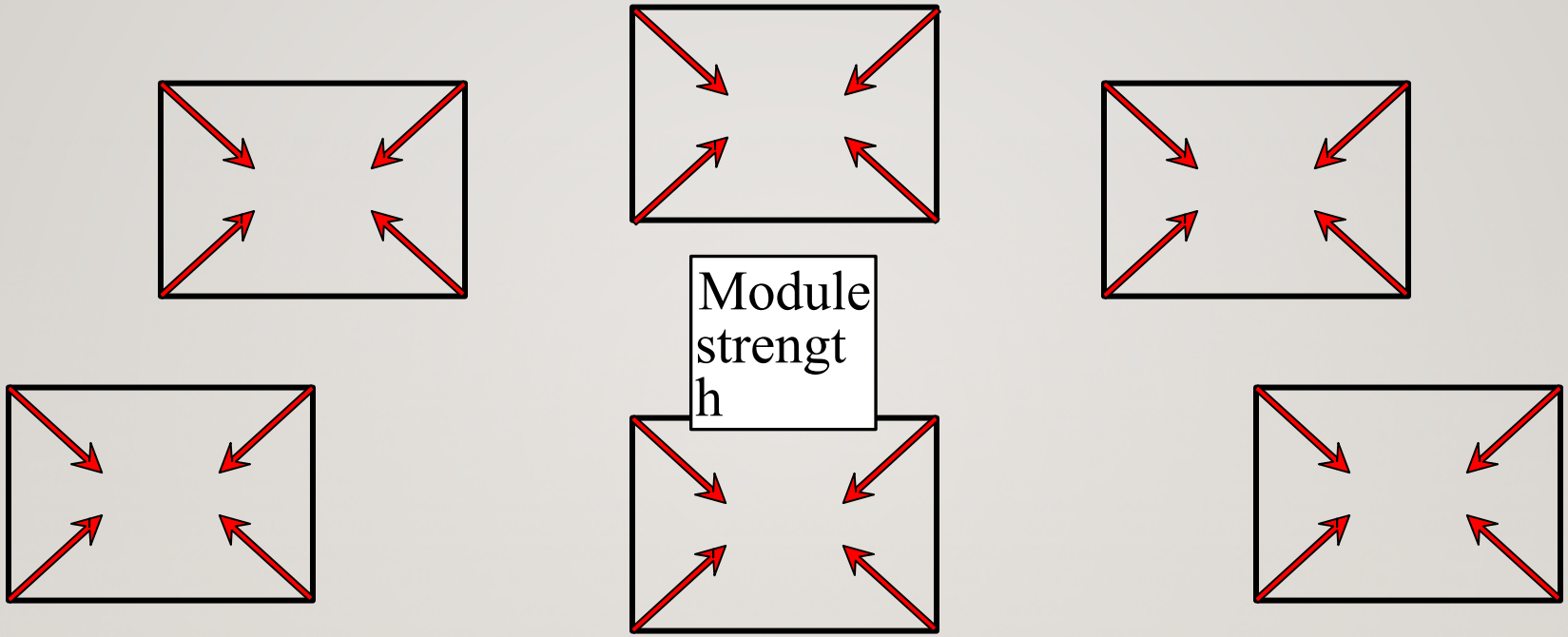


Fig: Cohesion=Strength of relations within modules

TYPES OF COHESION

Functional Cohesion	Best (high)
Sequential Cohesion	
Communicational Cohesion	
Procedural Cohesion	
Temporal Cohesion	
Logical Cohesion	
Coincidental Cohesion	Worst (low)

FUNCTIONAL COHESION

- Every essential element for a single computation is contained in the component.
- A and B are part of a single functional task. This is very good reason for them to be contained in the same procedure.
- A functional cohesion performs the task and functions.
- It is an ideal situation.

SEQUENTIAL COHESION

- An element outputs some data that becomes the input for other element, i.e., data flow between the parts.
- It occurs naturally in functional programming languages.

COMMUNICATIONAL COHESION

- Two elements operate on the same input data or contribute towards the same output data.
- Example- update record in the database and send it to the printer.

PROCEDURAL COHESION

- Procedural cohesion occurs in modules whose instructions although accomplish different task yet have been combined because there is a specific order in which the task are to be completed.
- **Elements of procedural cohesion ensure the order of execution.**
- Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.

TEMPORAL COHESION

- The elements are related by their timing involved.
- A module connected with temporal cohesion all the tasks must be executed in the same time span.
- This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.

LOGICAL COHESION

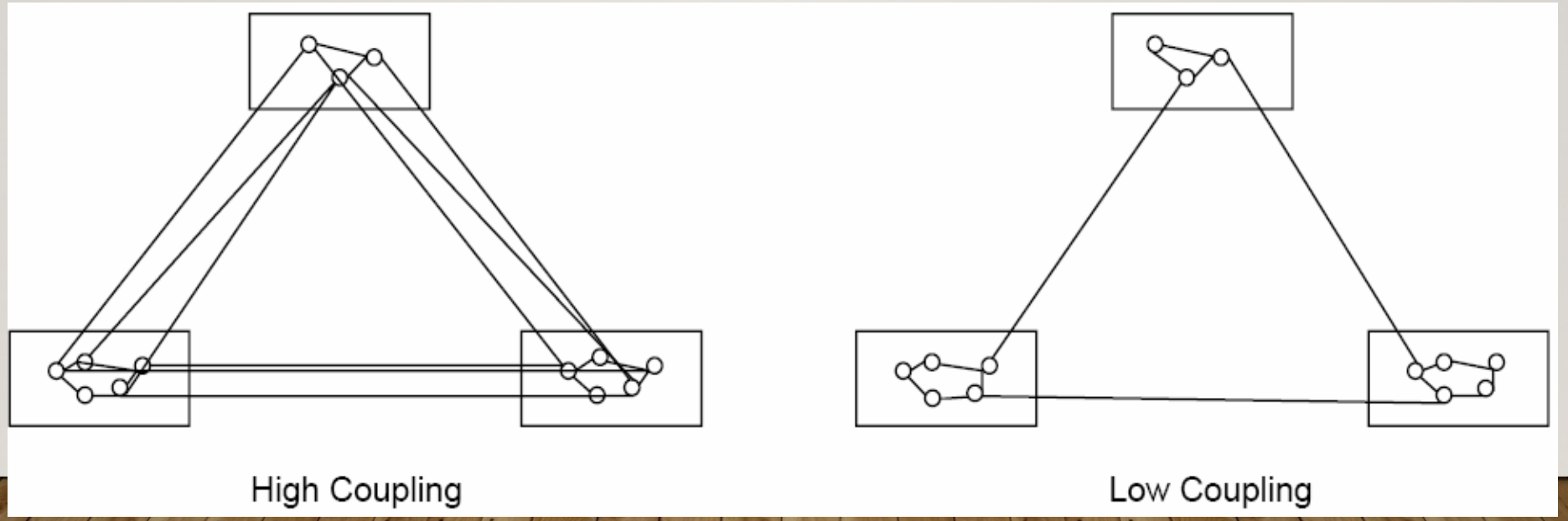
- The elements are logically related and not functionally.
- Ex- A component reads inputs from tape, disk, and network.
- All the code for these functions is in the same component.
- Operations are related, but the functions are significantly different.

COINCIDENTAL COHESION

- The elements are not related(unrelated).
- The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion.
- Ex- print next line and reverse the characters of a string in a single component.

RELATIONSHIP BETWEEN COHESION & COUPLING

- If software isn't modularized effectively, even small enhancements or changes can lead to project failure. Therefore, software engineers should design modules with the aim of achieving **high cohesion and low coupling.**

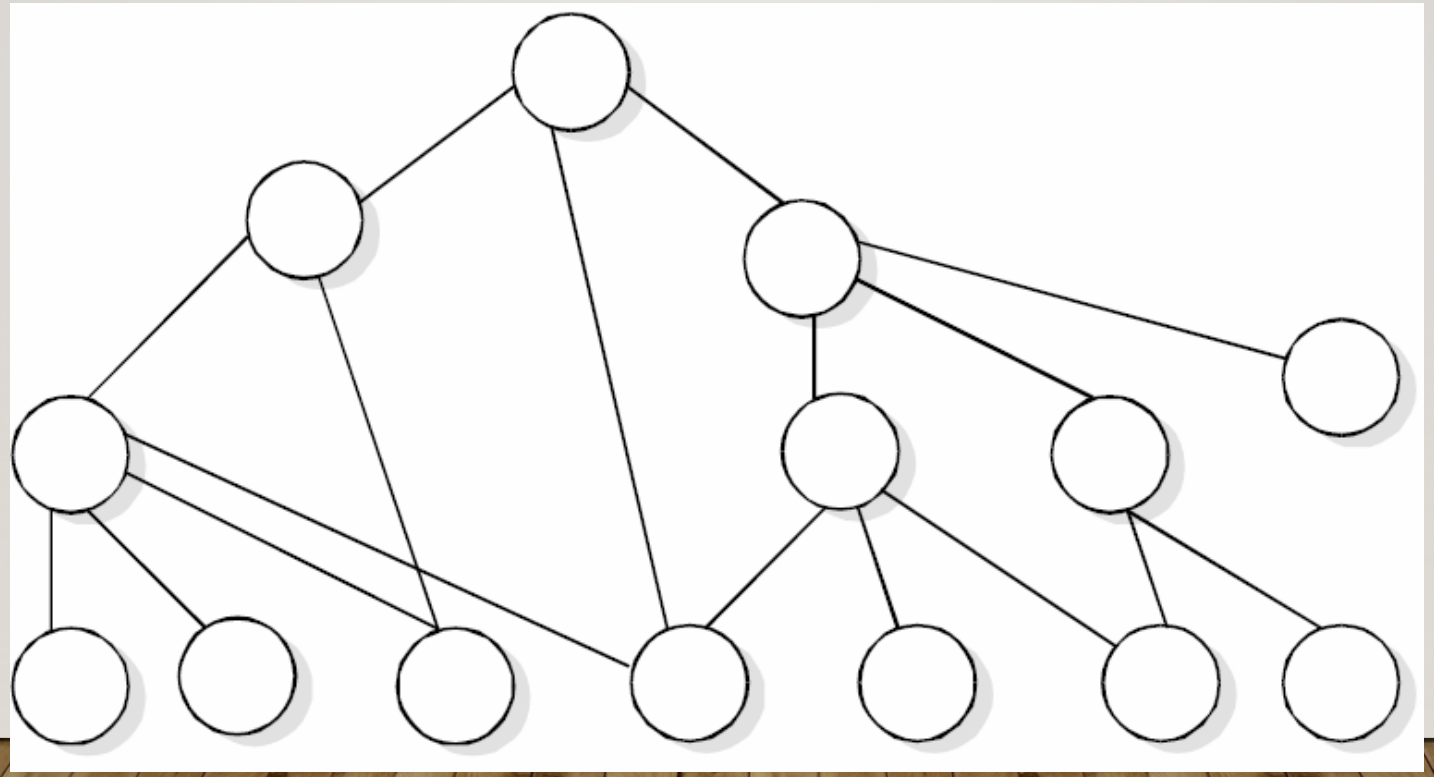


STRATEGY OF DESIGN

- **A good system design strategy is to organize the program modules in such a way that are easy to develop and latter to change.**
- Structured design techniques help developers to deal with the size and complexity of programs.
- Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program.
- It is important for two reasons:
  - First, even pre-existing code, if any, needs to be understood, organized and pieced together.
  - Second, it is still common for the project team to have to write some code and produce original programs that support the application logic of the system.

BOTTOM-UP DESIGN

- These modules are collected together in the form of a “library”.



TOP-DOWN DESIGN

- A top-down design approach starts by identifying the major modules of the system, decomposing them into their lower-level modules and iterating until the desired level of detail is achieved.
- This is stepwise refinement; starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly.

HYBRID DESIGN

- For top-down approach to be effective, some bottom-up approach is essential for the following reasons:
  - ➢To permit common sub modules.
  - ➢Near the bottom of the hierarchy, where the intuition is simpler, and the need for bottom-up testing is greater, because there are more number of modules at low levels than high levels.
  - ➢In the use of pre-written library modules, in particular, reuse of modules.

FUNCTION ORIENTED DESIGN

- The design process for software systems often has two levels. At the first level, the focus is on deciding which modules are needed for the system based on SRS (Software Requirement Specification) and how the modules should be interconnected.

- **Function Oriented Design is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function.**

Example:

Consider the example of scheme interpreter. Top-level function may look like:

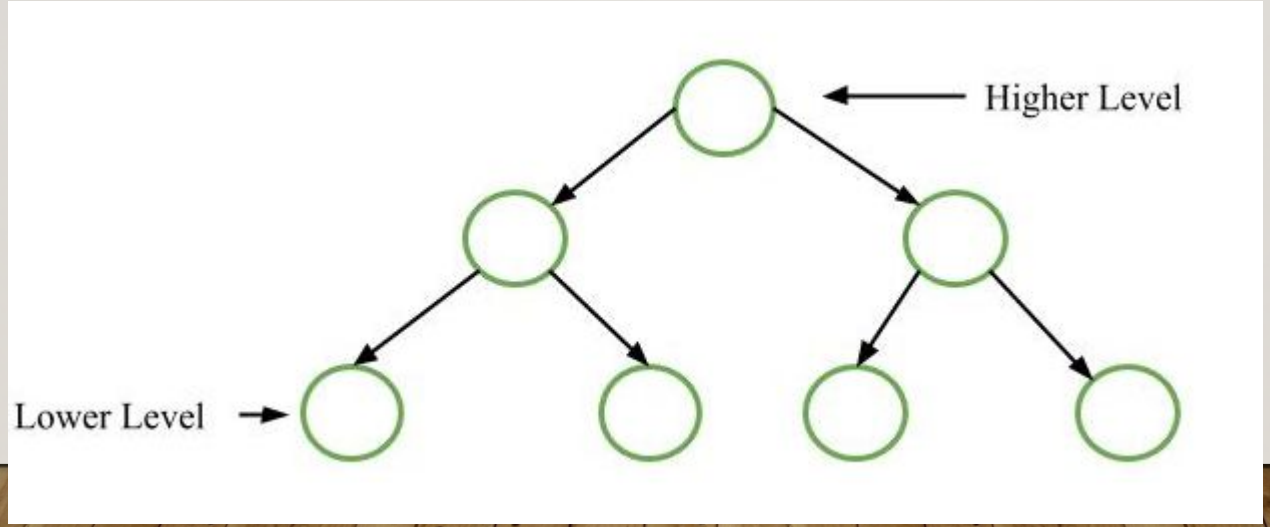
```
While (not finished)
{
        Read an expression from the terminal;
        Evaluate the expression;
        Print the value;
}
```

We thus get a fairly natural division of our interpreter into a "read" module, an "evaluate" module and a "print" module. Now we consider the "print" module and is given below:

```
        Print (expression exp)
{
        Switch (exp → type)
        Case integer: /*print an integer*/
        Case real:    /*print a real*/
        Case list:    /*print a list*/
        :::
}
```

GENERIC PROCEDURE

- Start with a high-level description of what the software/program does.
- Refine each part of the description by specifying in greater detail the functionality of each part.
- These points lead to a Top-Down Structure.



FUNCTION ORIENTED DESIGN NOTATIONS

- Design notations are largely meant to be used during the process of design and are used to represent design or design decisions.
- For a function-oriented design, the design can be represented graphically or mathematically by the following:

➢ **Data flow diagrams**

➢ **Data Dictionaries**

➢ **Structure Charts**

➢ **Pseudocode**

DATA FLOW DIAGRAMS(DFD)

- A data flow diagram (DFD) maps out the flow of information for any process or system.
- It uses defined symbols like rectangles, circles and arrows, plus short text labels, to show data inputs, outputs, storage points and the routes between each destination.

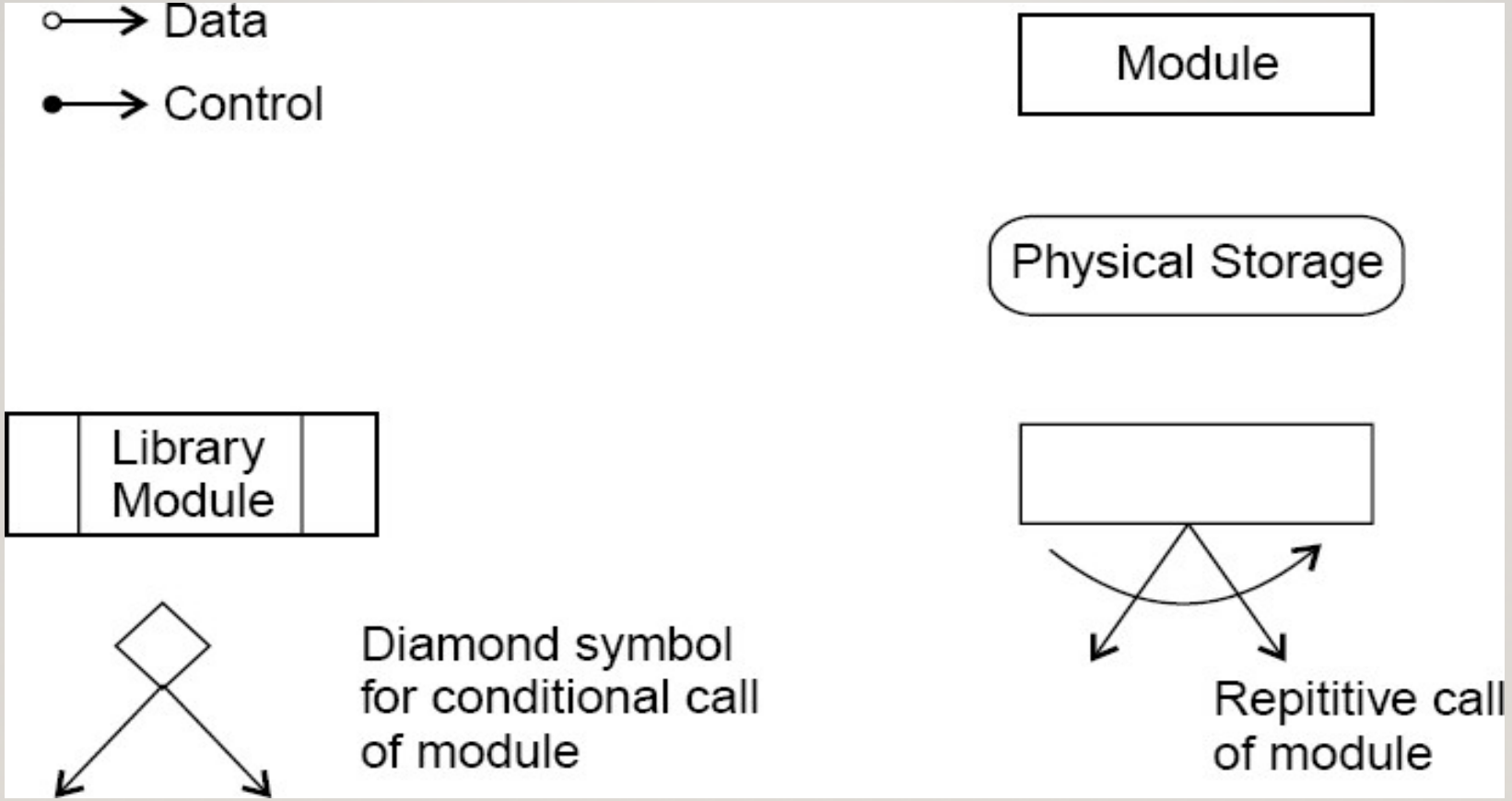
DATA DICTIONARIES

- Data dictionaries are simply repositories to store information about all data items defined in DFDs.
- At the requirement stage, data dictionaries contains data items.
- Data dictionaries include Name of the item, Aliases (Other names for items), Description / purpose, Related data items, Range of values, Data structure definition / form.

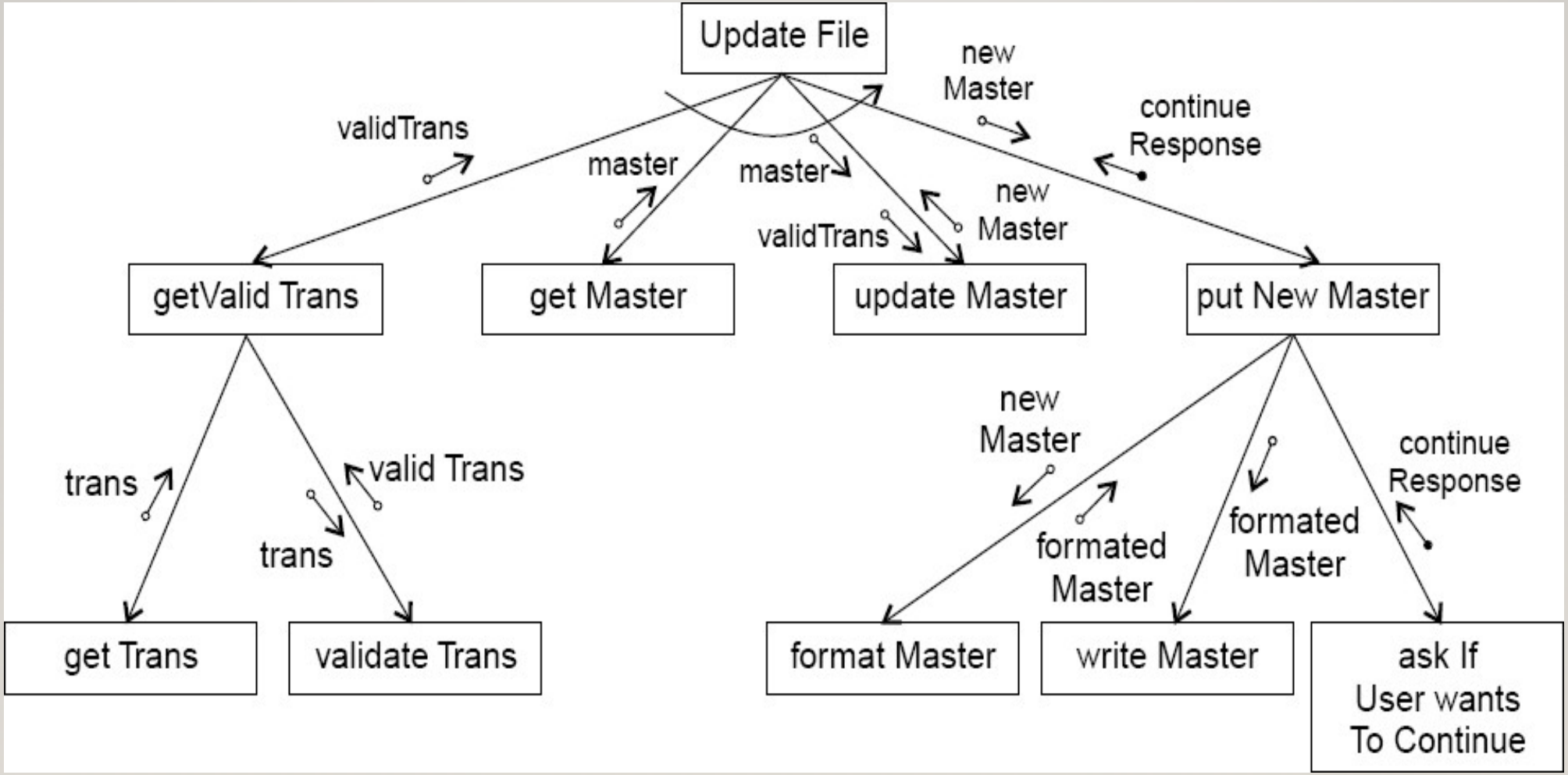
STRUCTURE CHARTS

- Structure chart is the hierarchical representation of system which partitions the system into black boxes (functionality is known to users, but inner details are unknown).
- Components are read from top to bottom and left to right.
- When a module calls another, it views the called module as a black box, passing required parameters and receiving results.
- Structure chart breaks down the entire system into the lowest functional modules and describes the functions and sub-functions of each module of a system in greater detail.

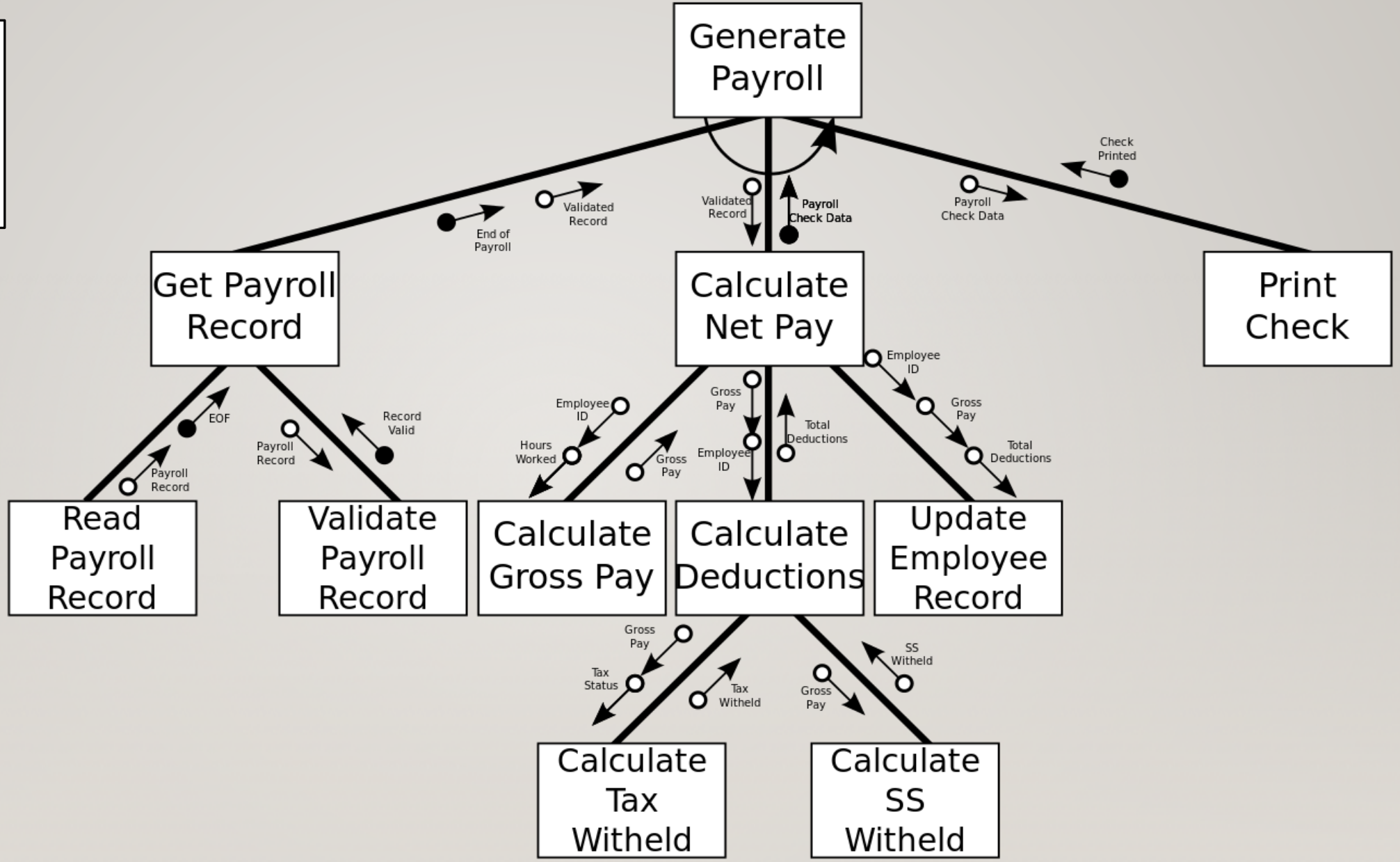
SYMBOLS IN STRUCTURE CHARTS



EXAMPLE FOR “UPDATE FILE”



EXAMPLE FOR “GENERATE PAYROLL”



PSEUDO CODE

- Pseudo Code is system description in short English like phrases describing the function. It uses keyword and indentation.
- Pseudocodes are used as replacement for flow charts. It decreases the amount of documentation required.
- **A Pseudocode is defined as a step-by-step description of an algorithm. Pseudocode does not use any programming language in its representation instead it uses the simple English language text such as It-Then-Else, While-Do, and End, as it is intended for human understanding rather than machine reading.**
- **Pseudocode is the intermediate state between an idea and its implementation(code) in a high-level language**

OBJECT ORIENTED DESIGN

- This methodology **employs object-oriented principles to model and design complex systems.** It involves analyzing the problem domain, representing it using objects and their interactions, and then designing a modular and scalable solution.
- Object Oriented Design begins with an examination of the real world “things” that are part of the problem to be solved. These things (which we will call objects) are characterized individually in terms of their attributes and behavior.
- It helps create systems that are easier to understand, maintain, and extend by organizing functionality into reusable and interconnected components.

OBJECTS

- The word “Object” is used as an **entity able to save a state (information) and which offers a number of operations (behavior) to either examine or affect this state.**
- An object is characterized by number of operations and a state which remembers the effect of these operations.

MESSAGES

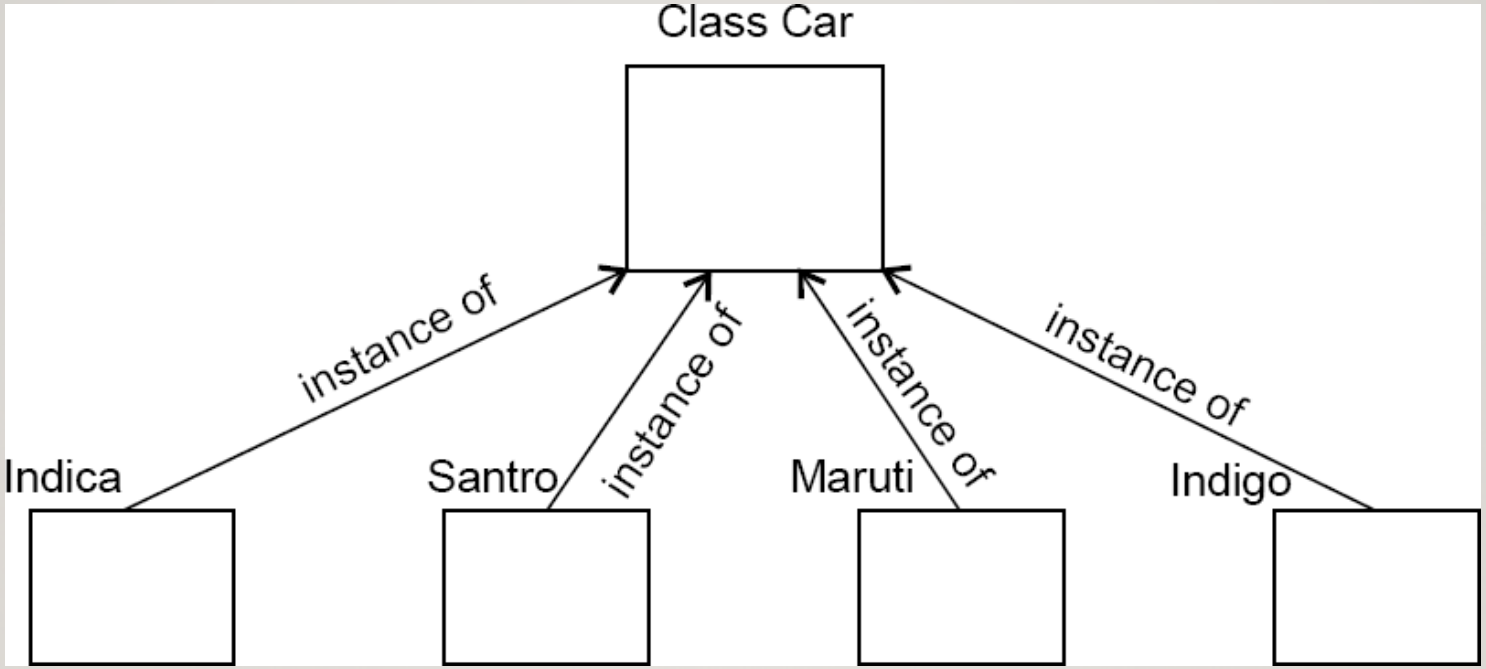
- Objects communicate by message passing.
- **Messages consist of the identity of the target object, the name of the requested operation and any other operation needed to perform the function.**
- Message are often implemented as procedure or function calls.

ABSTRACTION

- In object oriented design, complexity is managed using abstraction.
- **Abstraction is the elimination of the irrelevant and the amplification of the essentials.**

CLASS

- In any system, there shall be number of objects. Some of the objects may have common characteristics and we can group the objects according to these characteristics. This type of grouping is known as a class.
- Hence, **a class is a set of objects that share a common structure and a common behavior.**
- Classes are useful because they act as a blueprint for objects.



Indica, Santro, Maruti, Indigo are all instances of the class “car”

ATTRIBUTES

- An attributes is a data value held by the objects in a class.
- Each attributes has a value for each object instance.

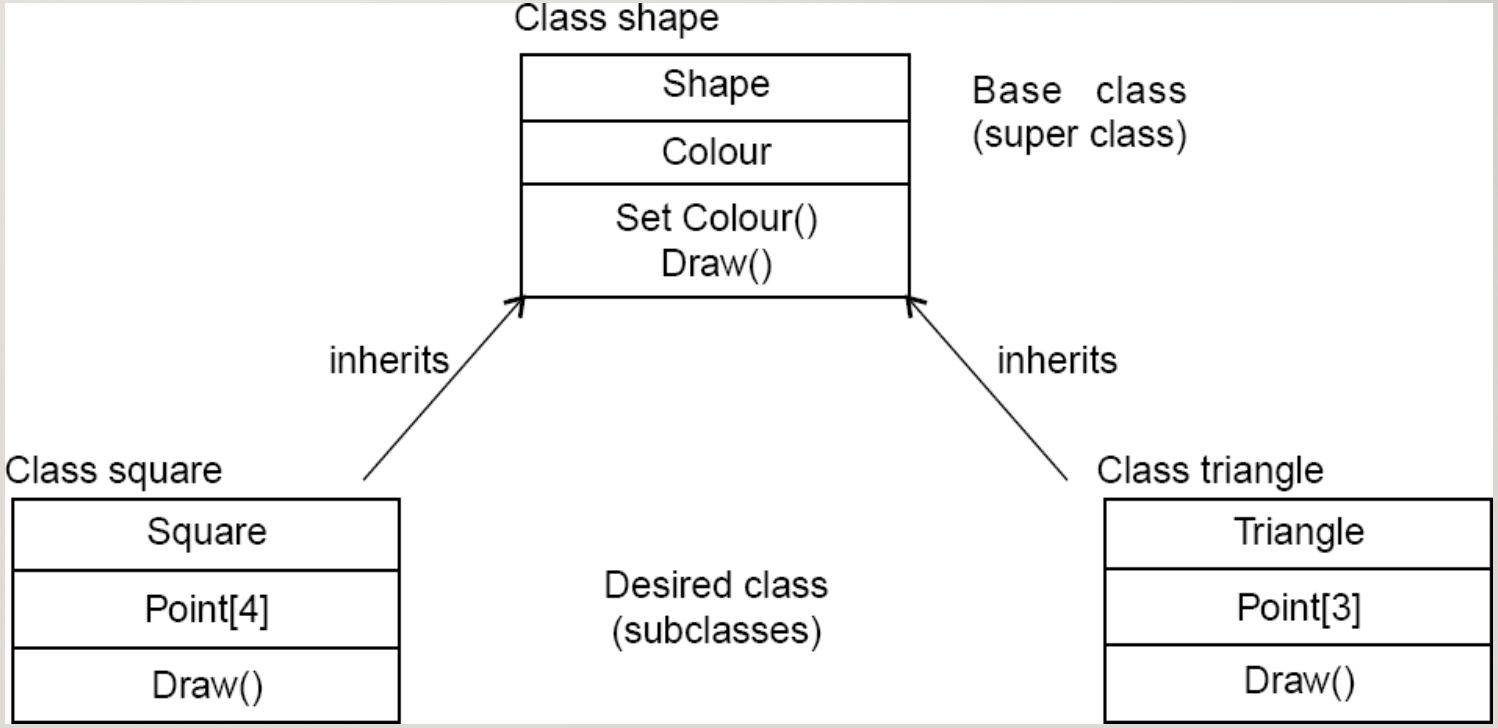
OPERATIONS

- **An operation is a function or transformation that may be applied to or by objects in a class.**
- All objects in a class share the same operations.

INHERITANCE

- **Abstracting common features in a new class.**

- The low level classes (known as subclasses or derived classes) inherit state and behavior from this high level class (known as a super class or base class).



POLYMORPHISM

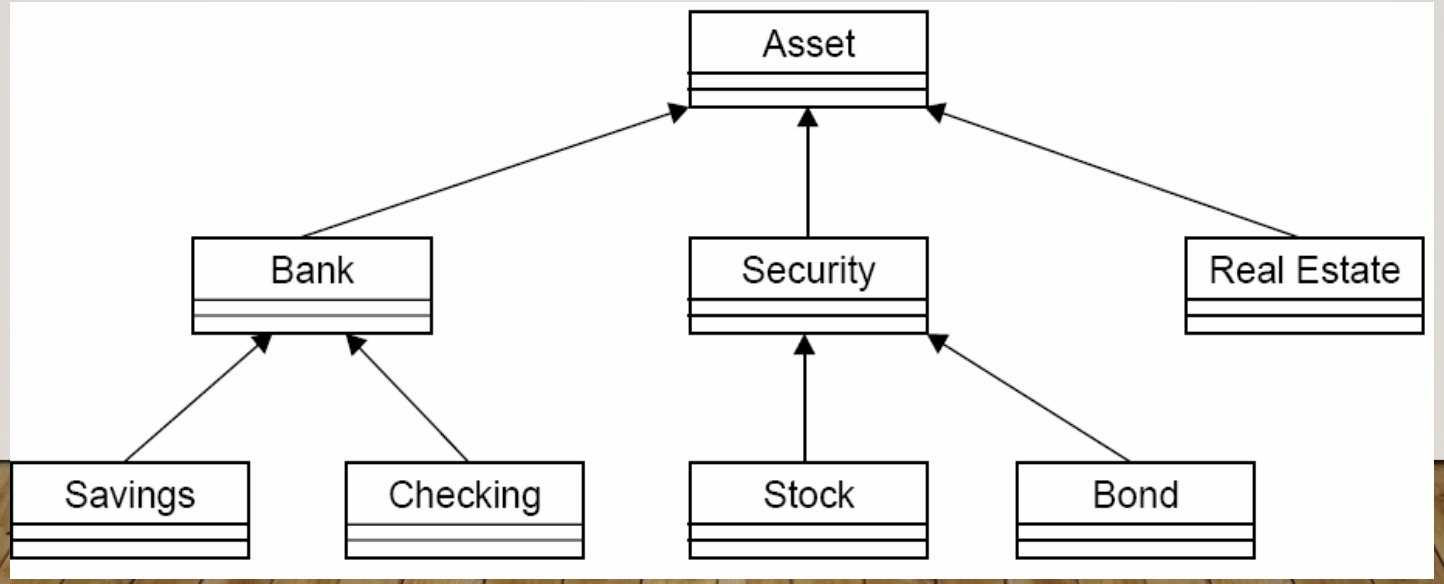
- When we abstract just the interface of an operation and leave the implementation to subclasses it is called a polymorphic operation and process is called polymorphism.

ENCAPSULATION

- Encapsulation is also commonly referred to as **“Information Hiding”.**
- It consists of the separation of the external aspects of an object from the internal implementation details of the object.

HIERARCHY

- Hierarchy involves organizing something according to some particular order or rank.
- It is another mechanism for reducing the complexity of software by being able to treat and express sub-types in a generic way.



SOFTWARE DESIGN DESCRIPTION(SDD)

- An SDD is a representation of a software system that is used as a medium for communicating software design information.
- **The purpose of an SDD is to show how the software system will be structured to satisfy the requirements identified in the SRS.**
- It is basically **the translation of requirements into a description of the software structure, software components, interfaces, and data necessary for the implementation phase.**
- Hence, SDD becomes the blue print for the implementation activity.

Contd.

- Design Description Information Content
  - ➢Introduction
  - ➢Design entities
  - ➢Design entity attributes
- The attributes and associated information items are defined in the following subsections:
  - ➢Identification
  - ➢Type
  - ➢Purpose
  - ➢Subordinates
  - ➢Dependencies
  - ➢Interface
  - ➢Resources
  - ➢Processing
  - ➢Data

FEW TERMINOLOGIES

- **Design entity:** An element (Component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.
- **Design View:** A subset of design entity attribute information that is specifically suited to the needs of a software project activity.
- **Entity attributes:** A named property or characteristics of a design entity. It provides a statement of fact about the entity.

SOFTWARE DESIGN DESCRIPTION (SDD) ORGANISATION

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions and acronyms
2. References
3. Decomposition description
   - 3.1 Module decomposition
     - 3.1.1 Module 1 description
     - 3.1.2 Module 2 description
   - 3.2 Concurrent Process decompostion
     - 3.2.1 Process 1 description
     - 3.2.2 Process 2 description
   - 3.3 Data decomposition
     - 3.3.1 Data entity 1 description
     - 3.3.2 Data entity 2 description

Design View	Scope	Entity attribute	Example representation
Decomposition description	Partition of the system into design entities	Identification, type purpose, function, subordinate	Hierarchical decomposition diagram, natural language
Dependency description	Description of relationships among entities of system resources	Identification, type, purpose, dependencies, resources	Structure chart, data flow diagrams, transaction diagrams
Interface description	List of everything a designer, developer, tester needs to know to use design entities that make up the system	Identification, function, interfaces	Interface files, parameter tables
Detail description	Description of the internal design details of an entity	Identification, processing, data	Flow charts, PDL etc.

WHAT IS CODING?

- The coding is the process of transforming the design of a system into a computer language format.
- This coding phase of software development is concerned with software translating design specification into the source code.
- It is necessary to write source code & internal documentation so that conformance of the code to its specification can be easily verified.
- Coding is done by the coder or programmers who are independent people than the designer.
- The goal is not to reduce the effort and cost of the coding phase, but to cut to the cost of a later stage. The cost of testing and maintenance can be significantly reduced with efficient coding.

GOALS OF CODING

- To translate the design of system into a computer language format.
- To reduce the cost of later phases.
- Making the program more readable.

CHARACTERISTICS OF PROGRAMMING LANGUAGE

Contd.

- **Readability:** A good high-level language will allow programs to be written in some methods that resemble a quite-English description of the underlying functions. The coding may be done in an essentially self-documenting way.
- **Portability:** High-level languages, being virtually machine-independent, should be easy to develop portable software.
- **Generality:** Most high-level languages allow the writing of a vast collection of programs, thus relieving the programmer of the need to develop into an expert in many diverse languages.
- **Brevity:** Language should have the ability to implement the algorithm with less amount of code. Programs mean in high-level languages are often significantly shorter than their low-level equivalents.

Contd.

- **Error checking:** A programmer is likely to make many errors in the development of a computer program. Many high-level languages invoke a lot of bugs checking both at compile-time and run-time.
- **Cost:** The ultimate cost of a programming language is a task of many of its characteristics.
- **Quick translation:** It should permit quick translation.
- **Efficiency:** It should authorize the creation of an efficient object code.
- **Modularity:** It is desirable that programs can be developed in the language as several separately compiled modules, with the appropriate structure for ensuring self-consistency among these modules.
- **Widely available:** Language should be widely available, and it should be feasible to provide translators for all the major machines and all the primary operating systems.

CODING STANDARDS

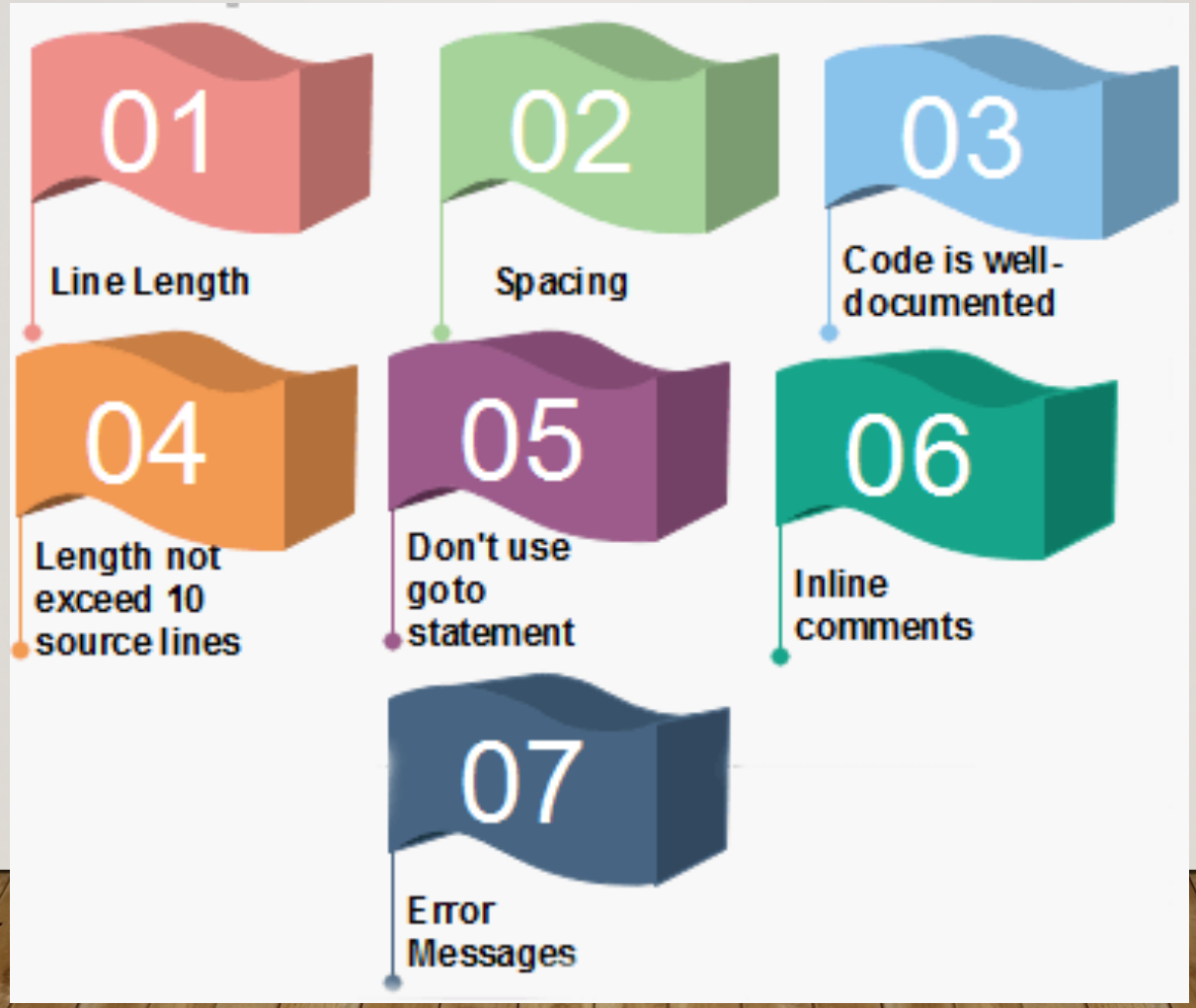
Contd.

- **Indentation:** Proper and consistent indentation is essential in producing easy to read and maintainable programs. Indentation should be used to:
  - Emphasize the body of a control structure such as a loop or a select statement.
  - Emphasize the body of a conditional statement
  - Emphasize a new scope block
- **Inline comments:** Inline comments analyze the functioning of the subroutine, or key aspects of the algorithm shall be frequently used.
- **Rules for limiting the use of global:** These rules file what types of data can be declared global and what cannot.

Contd.

- **Structured Programming:** Structured (or Modular) Programming methods shall be used.
- **Naming conventions for global variables, local variables, and constant identifiers:** A possible naming convention can be that global variable names always begin with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
- **Error return conventions and exception handling system:** Different functions in a program report the way error conditions are handled should be standard within an organization. For example, different tasks while encountering an error condition should either return a 0 or 1 consistently.

CODING GUIDELINES



Contd.

- **Line Length:** It is considered a good practice to keep the length of source code lines at or below 80 characters. Lines longer than this may not be visible properly on some terminals and tools. Some printers will truncate lines longer than 80 columns.
- **Spacing:** The appropriate use of spaces within a line of code can improve readability.

  Example:

  Bad:
  ```
  cost=price+(price*sales_tax)
  fprintf(stdout ,"The total cost is %5.2f\n",cost);
  ```

  Better:
  ```
  cost = price + ( price * sales_tax )
  fprintf (stdout,"The total cost is %5.2f\n",cost);
  ```

Contd.

- **The code should be well-documented:** As a rule of thumb, there must be at least one comment line on the average for every three-source line.
- **The length of any function should not exceed 10 source lines:** A very lengthy function is generally very difficult to understand as it possibly carries out many various functions. For the same reason, lengthy functions are possible to have a disproportionately larger number of bugs.
- **Do not use goto statements:** Use of goto statements makes a program unstructured and very tough to understand.
- **Inline Comments:** Inline comments promote readability.
- **Error Messages:** Error handling is an essential aspect of computer programming. This does not only include adding the necessary logic to test for and handle errors but also involves making error messages meaningful.

E-BOOKS

- **Software Engineering- A Practitioner's Approach** by **Roger S. Pressman**
- **Software Engineering** Ninth Edition by **Ian Sommerville**
- **Software Engineering** Third Edition by **K.K Aggarwal & Yogesh Singh**