Software Engineering

Dr. Mritunjay Shall Peelam
Assistant Professor-SG, UPES

Course Introduction and Objectives

Course Code	Course name	L	T	P	C
CSEG2064	Software Engineering	3	0	0	3
Total Units to be Covered: 5		Total Contact Hours: 45			
Prerequisite(s):		Syllabus version: 1.0			

Course Objectives

1. To explore software development methodologies (waterfall, agile, DevOps) and their integration of testing, quality assurance, reliability, and risk management.
2. To comprehend software requirements engineering and develop skills in creating well-structured Software Requirements Specifications (SRS).
3. To acquire understanding of planning a software project, its cost estimation models and to understand the software quality models.

Course Introduction and Objectives

Course Outcomes

CO 1. Understand the fundamental concepts and importance of Software Engineering in modern software development.

CO 2. Learn various software development methodologies, including Agile, Waterfall, and iterative approaches.

CO 3. Explore software design principles and architectural patterns for creating robust and maintainable software systems.

CO 4. Apply project management principles to effectively plan, monitor, and control software projects.

Course Introduction and Objectives

Syllabus

Unit I: Introduction to Software Engineering **7 Lecture Hours**

Definition of Software Engineering, S/W characteristics, applications, Software development life cycle ; Life Cycle Models – Waterfall (classical and iterative), Spiral, Prototyping & RAD Models, Software processes, Process Models – overview Agile Model and Various Agile methodologies - Scrum, XP, Lean, and Kanban. Scope of each model and their comparison in real-world case studies.

Course Introduction and Objectives

Unit II: Requirements Modelling and Design **9 Lecture Hours**

System and software requirements; Requirements Engineering-Crucial steps; types of requirements, Functional and non-functional requirements; Domain requirements; User requirements; Elicitation and analysis of requirements; Requirements documentation – Nature of Software, Software requirements specification, Use case diagrams with guidelines, DFD, SRS Structure, SRS Case study, Design concepts and principles - Abstraction - Refinement - Modularity Cohesion coupling, Architectural design, Detailed Design Transaction Transformation, Refactoring of designs, Object-oriented Design User-Interface Design.

Course Introduction and Objectives

Unit III: Software Reliability **9 Lecture Hours**

Introduction to Software Reliability; Hardware reliability vs. Software reliability; Reliability metrics; Failure and Faults – Prevention, Removal, Tolerance, Forecast; Dependability Concept – Failure Behavior, Characteristics, Maintenance Policy; Reliability and Availability Modeling; Reliability Evaluation Testing methods, Limits, Starvation, Coverage, Filtering; Microscopic Model of Software Risk; Classes of software reliability Models; Statistical reliability models; Reliability growth models; Defining and interpreting reliability metrics; Fault Detection and Prevention; Techniques for detecting and mitigating software faults; Static analysis tools and techniques; Dynamic analysis methods; Software Fault Tolerance; Software Maintenance and Reliability; Reliability Assessment and Evaluation; Methods for assessing and quantifying software reliability; Case Studies and Real-world Applications.

Course Introduction and Objectives

Unit IV: Software Testing, metrics and Quality Assurance **10 Lecture Hours**

Testing types and techniques such as black box, white box, and gray box testing, functional and structural testing; Test-driven development, code coverage, and quality metrics; Testing process, design of Test cases, testing techniques - boundary value analysis - equivalence class testing - decision table testing, cause-effect graphing, path testing, data flow testing, and mutation testing. Unit, integration, system, alpha, and beta testing, debugging techniques; verification and validation techniques, levels of testing, regression testing, quality management activities, product and process quality standards (ISO9000, CMM), metrics understanding (process, product, project metrics), size metrics (LOC, Function Count, Albrecht FPA), product metrics, metrics for software maintenance, cost estimation techniques (static, single variable, multivariable models), cost-benefit evaluation techniques, Testing tools and standards such as Jira and Selenium, test automation frameworks and tools (Selenium, Appium, JUnit), performance testing and load testing, and defect management and root cause analysis.

Course Introduction and Objectives

Unit V: Software Quality and Risk Management **10 Lecture Hours**

McCall quality factors, ISO and CMM Model, Tools and Techniques for Quality Control, Pareto Analysis, Statistical Sampling, Quality Control Charts and the seven Run Rule. Modern Quality Management, Risk Management – importance, types, process and phases, qualitative and quantitative risk analysis, Risk Analysis and Assessment, Risk Strategies, Risk Monitoring and Control, Risk Response and Evaluation. Software Reliability: Reliability Metrics, Reliability Growth Modeling. Use Case: Defect Tracking and Management. Test Automation Tools: Jira, Selenium:, Appium; JUnit.

Text Books and References

Textbooks	1. Roger S. Pressman, "Software Engineering: A practitioner's approach", 7th Edition, McGraw Hill, 2009. 2. Pankaj Jalote, "An integrated approach to Software Engineering", 3rd Edition, Springer/Narosa, 2005.
Reference books	1. James F. Peters, and Witold Pedrycz, "Software Engineering: an Engineering approach", John Wiley, 2007. 2. Waman S Jawadekar, "Software Engineering principles and practice", McGraw Hill, 2004.
Web Resources **Journals**	
MOOCs, online courses	

Modes of Evaluation

Modes of Evaluation: Quiz/Assignment/ presentation/ extempore/ Written Examination etc.

Examination Scheme

Components	IA	MID SEM	End Sem	Total
Weightage (%)	50	20	30	100

Unit-2

Unit II: Requirements Modelling and Design **9 Lecture Hours**

System and software requirements; Requirements Engineering-Crucial steps; types of requirements, Functional and non-functional requirements; Domain requirements; User requirements; Elicitation and analysis of requirements; Requirements documentation – Nature of Software, Software requirements specification, Use case diagrams with guidelines, DFD, SRS Structure, SRS Case study, Design concepts and principles - Abstraction - Refinement - Modularity Cohesion coupling, Architectural design, Detailed Design Transaction Transformation, Refactoring of designs, Object-oriented Design User-Interface Design.

Requirements describe

What not How

Produces one large document written in natural language contains a description of what the system will do without describing how it will do it.

Crucial process steps

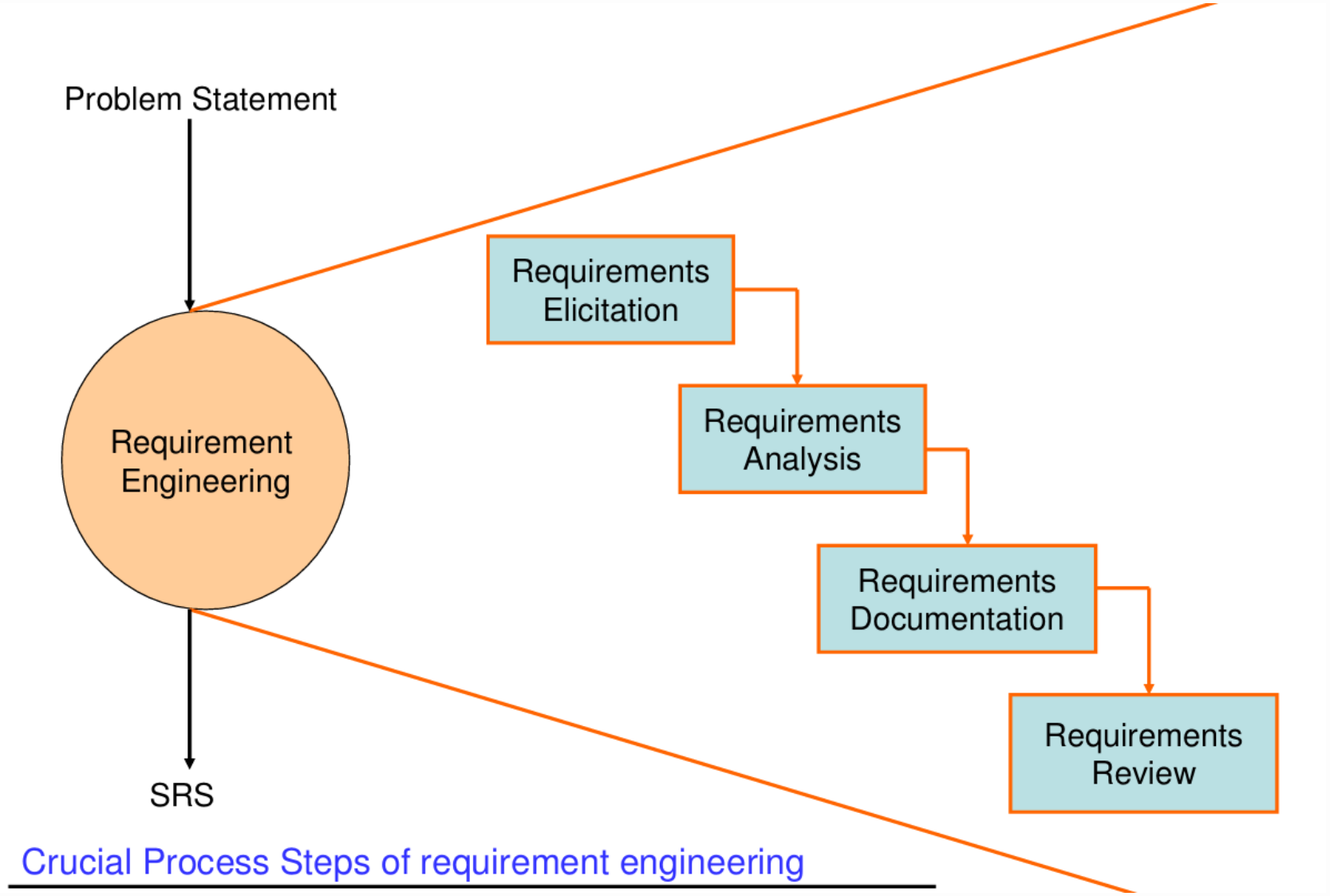
Quality of product → Process that creates it

Without well written document

-- Developers do not know what to build

-- Customers do not know what to expect

-- What to validate



Crucial Process Steps of requirement engineering

Requirement Engineering

Requirement Engineering is the disciplined application of proven principles, methods, tools, and notations to describe a proposed system's intended behavior and its associated constraints.

SRS may act as a contract between developer and customer.

State of practice

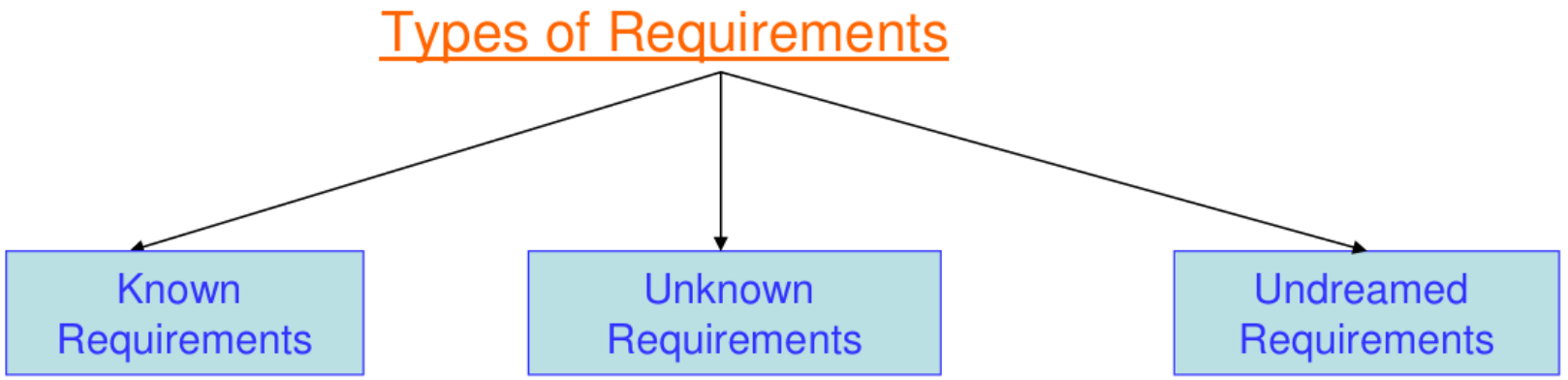
Requirements are difficult to uncover

- Requirements change
- Over reliance on CASE Tools
- Tight project Schedule
- Communication barriers
- Market driven software development
- Lack of resources

Example

A University wish to develop a software system for the student result management of its M.Tech. Programme. A problem statement is to be prepared for the software development company. The problem statement may give an overview of the existing system and broad expectations from the new software system.

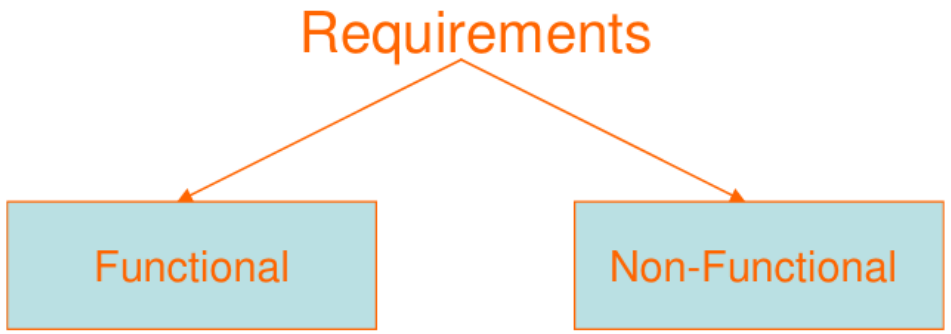
Types of Requirements



Stakeholder: Anyone who should have some direct or indirect influence on the system requirements.

--- User

--- Affected persons



Types of Requirements

Functional requirements describe what the software has to do. They are often called product features.

Non Functional requirements are mostly quality requirements. That stipulate how well the software does, what it has to do.

- Availability
- Reliability
- Usability
- Flexibility

} For Users

- Maintainability
- Portability
- Testability

} For Developers

Types of Requirements

User and system requirements

- User requirement are written for the users and include functional and non functional requirement.
- System requirement are derived from user requirement.
- The user system requirements are the parts of software requirement and specification (SRS) document.

Types of Requirements

Interface Specification

- Important for the customers.

TYPES OF INTERFACES

- Procedural interfaces (also called Application Programming Interfaces (APIs)).
- Data structures
- Representation of data.

Requirements Elicitation

Perhaps

- Most difficult
- Most critical
- Most error prone
- Most communication intensive

Succeed

effective customer developer partnership

Selection of any method

1. It is the only method that we know
2. It is our favorite method for all situations
3. We understand intuitively that the method is effective in the present circumstances.

Normally we rely on first two reasons.

Requirements Elicitation

1. Interviews

Both parties have a common goal

--- open ended
--- structured

} Interview

Success of the project

Selection of stakeholder

1. Entry level personnel
2. Middle level stakeholder
3. Managers
4. Users of the software (Most important)

Requirements Elicitation

Types of questions.

- Any problems with existing system
- Any Calculation errors
- Possible reasons for malfunctioning
- No. of Student Enrolled

Requirements Elicitation

5. Possible benefits
6. Satisfied with current policies
7.How are you maintaining the records of previous students?
8. Any requirement of data from other system
9. Any specific problems
10. Any additional functionality
11. Most important goal of the proposed development

At the end, we may have wide variety of expectations from the proposed software.

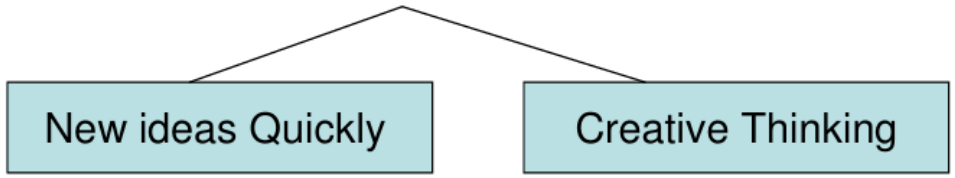
Requirements Elicitation

2. Brainstorming Sessions

It is a group technique

↓

group discussions



Prepare long list of requirements

Categorized
Prioritized
Pruned

*Idea is to generate views ,not to vet them.

Groups

1. Users 2. Middle Level managers 3. Total Stakeholders

Requirements Elicitation

A Facilitator may handle group bias, conflicts carefully.

-- Facilitator may follow a published agenda

-- Every idea will be documented in a way that everyone can see it.

--A detailed report is prepared.

3. Facilitated Application specification Techniques (FAST)

-- Similar to brainstorming sessions.

-- Team oriented approach

-- Creation of joint team of customers and developers.

Requirements Elicitation

Guidelines

1. Arrange a meeting at a neutral site.
2. Establish rules for participation.
3. Informal agenda to encourage free flow of ideas.
4. Appoint a facilitator.
5. Prepare definition mechanism board, worksheets, wall stickier.
6. Participants should not criticize or debate.

FAST session Preparations

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

Requirements Elicitation

1. Part of environment that surrounds the system.
2. Produced by the system.
3. Used by the system.

A. List of constraints
B. Functions
C. Performance criteria

Activities of FAST session

1. Every participant presents his/her list
2. Combine list for each topic
3. Discussion
4. Consensus list
5. Sub teams for mini specifications
6. Presentations of mini-specifications
7. Validation criteria
8. A sub team to draft specifications

Requirements Elicitation

4. Quality Function Deployment

-- Incorporate voice of the customer

Technical requirements.

Documented

Prime concern is customer satisfaction

What is important for customer?

-- Normal requirements

-- Expected requirements

-- Exciting requirements

Requirements Elicitation

Steps

1. Identify stakeholders
2. List out requirements
3. Degree of importance to each requirement.

Requirements Elicitation

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

Requirement Engineer may categorize like:

(i) It is possible to achieve
(ii) It should be deferred & Why
(iii) It is impossible and should be dropped from consideration

First Category requirements will be implemented as per priority assigned with every requirement.

5. The Use Case Approach

Ivar Jacobson & others introduced Use Case approach for elicitation & modeling.

Use Case – give functional view

The terms

- Use Case
- Use Case Scenario
- Use Case Diagram

} Often Interchanged

But they are different

Use Cases are structured outline or template for the description of user requirements modeled in a structured language like English.

Requirements Elicitation

Use case Scenarios are unstructured descriptions of user requirements.

Use case diagrams are graphical representations that

may be decomposed into further levels of abstraction.

Components of Use Case approach

Actor:

An actor or external agent, lies outside the system model, but interacts with it in some way.

Actor → Person, machine, information System

Requirements Elicitation

- Cockburn distinguishes between Primary and secondary actors.
- A Primary actor is one having a goal requiring the assistance of the system.
- A Secondary actor is one from which System needs assistance.

Use Cases

A use case is initiated by a user with a particular goal in mind, and completes successfully when that goal is satisfied.

Requirements Elicitation

* It describes the sequence of interactions between actors and the system necessary to deliver the services that satisfies the goal.
* Alternate sequence
* System is treated as black box.

Thus

Use Case captures who (actor) does what (interaction) with the system, for what purpose (goal), without dealing with system internals.

Requirements Elicitation

*defines all behavior required of the system, bounding the scope of the system.

Jacobson & others proposed a template for writing Use cases as shown below:

1. Introduction

Describe a quick background of the use case.

2.Actors

List the actors that interact and participate in the use cases.

3.Pre Conditions

Pre conditions that need to be satisfied for the use case to perform.

4. Post Conditions

Define the different states in which we expect the system to be in, after the use case executes.

Requirements Elicitation

5. Flow of events

5.1 Basic Flow
List the primary events that will occur when this use case is executed.

5.2 Alternative Flows
Any Subsidiary events that can occur in the use case should be separately listed. List each such event as an alternative flow.
A use case can have many alternative flows as required.

6.Special Requirements

Business rules should be listed for basic & information flows as special requirements in the use case narration .These rules will also be used for writing test cases. Both success and failures scenarios should be described.

7.Use Case relationships

For Complex systems it is recommended to document the relationships between use cases. Listing the relationships between use cases also provides a mechanism for traceability

Functional Requirements	Non-Functional Requirements
Defines a system or its component	Defines a system's quality attributes
Specifies what a system should do	Limits or manages how a system should do what it should
User specifies the requirements	Technical team specifies the requirements
Is a mandatory requirement	Isn't a mandatory requirement
Use case captures the requirements	Quality attributes capture the requirements
Defined to a component level	Defines the whole system
Helps to verify software functionality	Helps to verify software performance
Functional system, integration, API and end-to-end testing	Non-functional performance, security and stress testing
Defines easily	Is harder to define
Focuses on user requirements	Focuses on user expectations
Defines product features	Defines product properties

Functional Requirements Examples

- Some software features and functions examples of functional requirements include:
- System emails are sent each time an order is placed on the website or app.
- The ability to subscribe to a newsletter with an email.
- Site users use their mobile numbers to verify accounts.
- The feature to enter a username and password to authenticate a login.
- A button to report a system problem.
- CRUD users can change, read, update, or delete account information.
- Adjust cart items while shopping online, or check them out.
- A filter to reduce results in a search engine.
- A function to download or print a page from the system.
- The function that allows users to authenticate accounts with external platforms.
- An app sends updates, promotional content, or reminders to users.

Non-Functional Requirements Examples

- How fast the system responds to input commands and increasing system loads.
- Security measures, including password generators, security questions, and account locking.
- The system's portability across multiple devices and platforms.
- Whether the system is compatible with other applications running on the same device.
- The amount of capacity a system has to store new settings and preferences.
- A system's reliability related to the probability of failure and the number of and time between critical failures.
- How the system responds to external interfaces and environmental changes.
- How the system localises languages, geographic locations, time zones, and currency.
- Criteria that manage features like navigation, performance quality, and tool purposes.

1. Which one of the following is a functional requirement ?

a) Maintainability

b) Portability

c) Robustness

d) None of the mentioned

1. Which one of the following is a functional requirement ?

a) Maintainability

b) Portability

c) Robustness

d) None of the mentioned

2. “Consider a system where, a heat sensor detects an intrusion and alerts the security company.” What kind of a requirement the system is providing ?

a) Functional

b) Non-Functional

c) Known Requirement

d) None of the mentioned

2. “Consider a system where, a heat sensor detects an intrusion and alerts the security company.” What kind of a requirement the system is providing ?

a) Functional

b) Non-Functional

c) Known Requirement

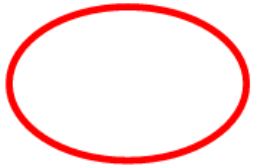
d) None of the mentioned

Use case Diagrams

-- represents what happens when actor interacts with a system.

-- captures functional aspect of the system.

Actor



Use Case

Relationship between actors and use case and/or between the use cases.

-- Actors appear outside the rectangle.

--Use cases within rectangle providing functionality.

--Relationship association is a solid line between actor & use cases.

Requirements Elicitation

*Use cases should not be used to capture all the details of the system.

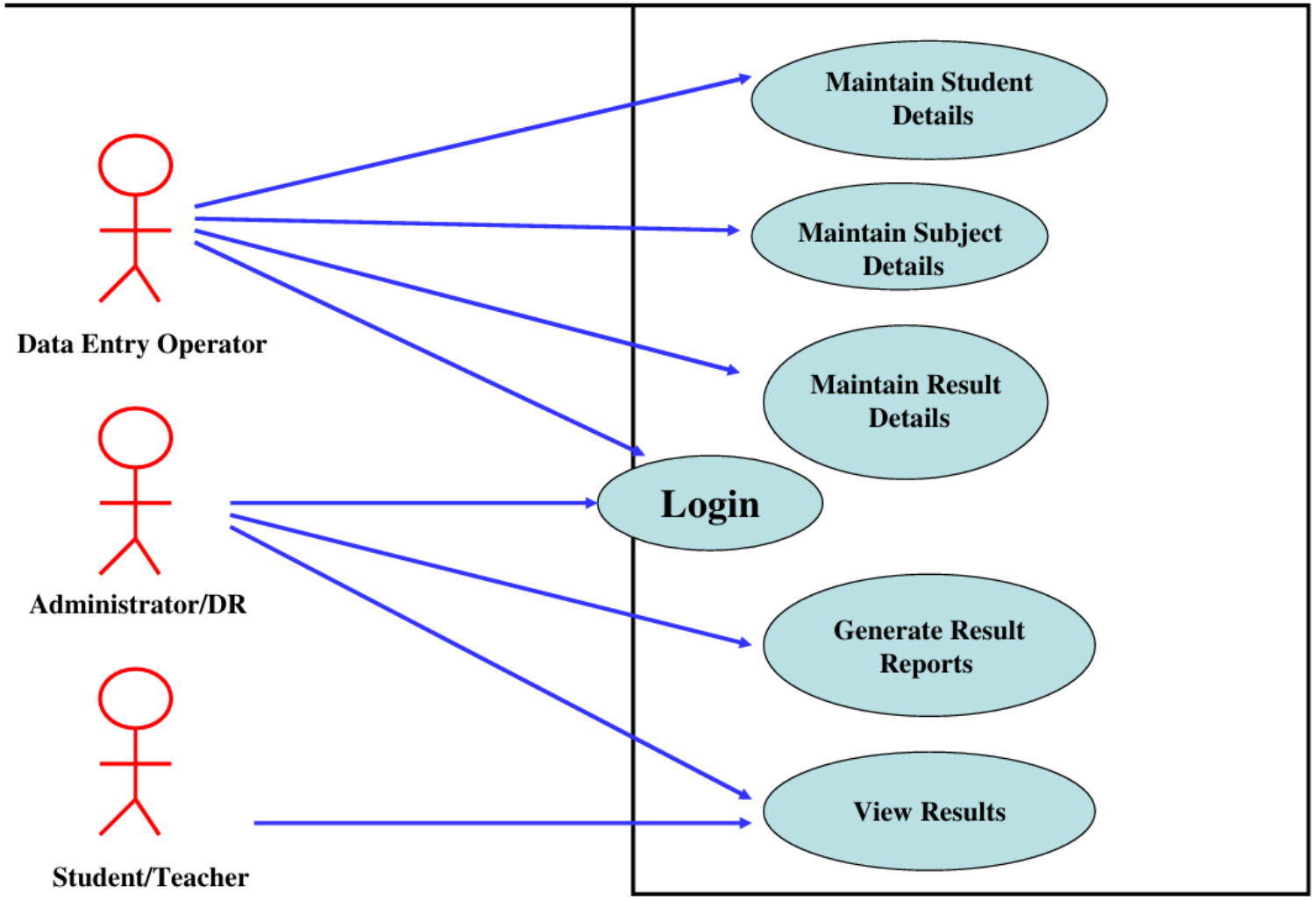
*Only significant aspects of the required functionality

*No design issues

*Use Cases are for “what” the system is , not “how” the system will be designed

* Free of design characteristics

Use case diagram for Result Management System



Requirements Elicitation

1. Maintain student Details

   Add/Modify/update students details like name, address.

2.Maintain subject Details

   Add/Modify/Update Subject information semester wise

3.Maintain Result Details

   Include entry of marks and assignment of credit points for each paper.

4. Login

   Use to Provide way to enter through user id & password.

5. Generate Result Report

   Use to print various reports

6. View Result
   (i) According to course code
   (ii) According to Enrollment number/roll number

Requirements Elicitation (Use Case)

Login

1.1 Introduction : This use case describes how a user logs into the Result Management System.

1.2 Actors : (i) Data Entry Operator
(ii) Administrator/Deputy Registrar

1.3 Pre Conditions : None

1.4 Post Conditions : If the use case is successful, the actor is logged into the system. If not, the system state is unchanged.

Requirements Elicitation (Use Case)

1.5 Basic Flow : This use case starts when the actor wishes to login to the Result Management system.

(i) System requests that the actor enter his/her name and password.

(ii) The actor enters his/her name & password.

(iii) System validates name & password, and if finds correct allow the actor to logs into the system.

1.6 Alternate Flows

1.6.1 Invalid name & password

If in the basic flow, the actor enters an invalid name and/or password, the system displays an error message. The actor can choose to either return to the beginning of the basic flow or cancel the login, at that point, the use case ends.

1.7 Special Requirements:

None

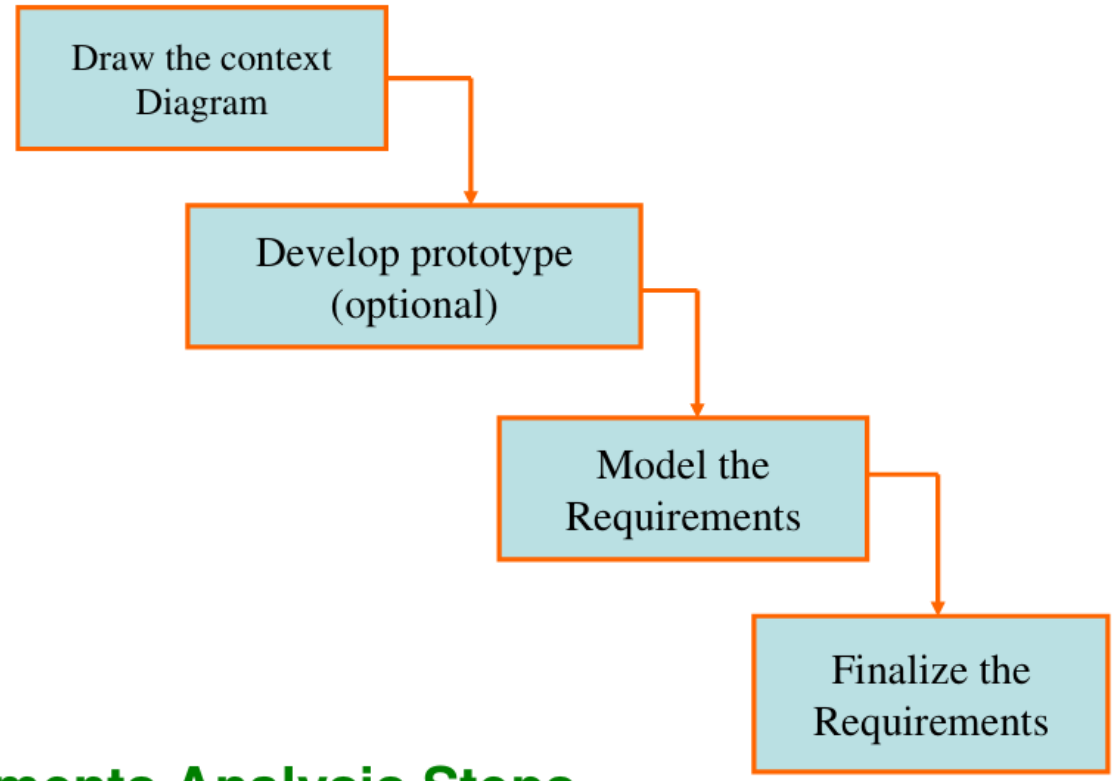
1.8 Use case Relationships:

None

Requirements Analysis

We analyze, refine and scrutinize requirements to make consistent & unambiguous requirements.

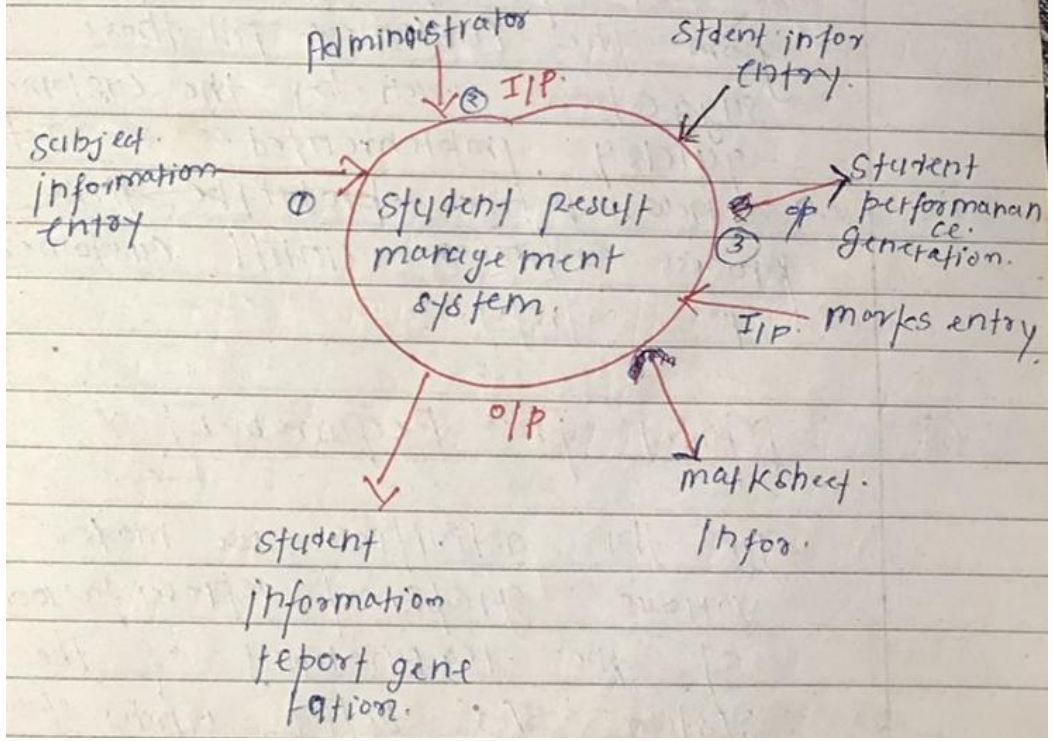
Steps



Requirements Analysis Steps

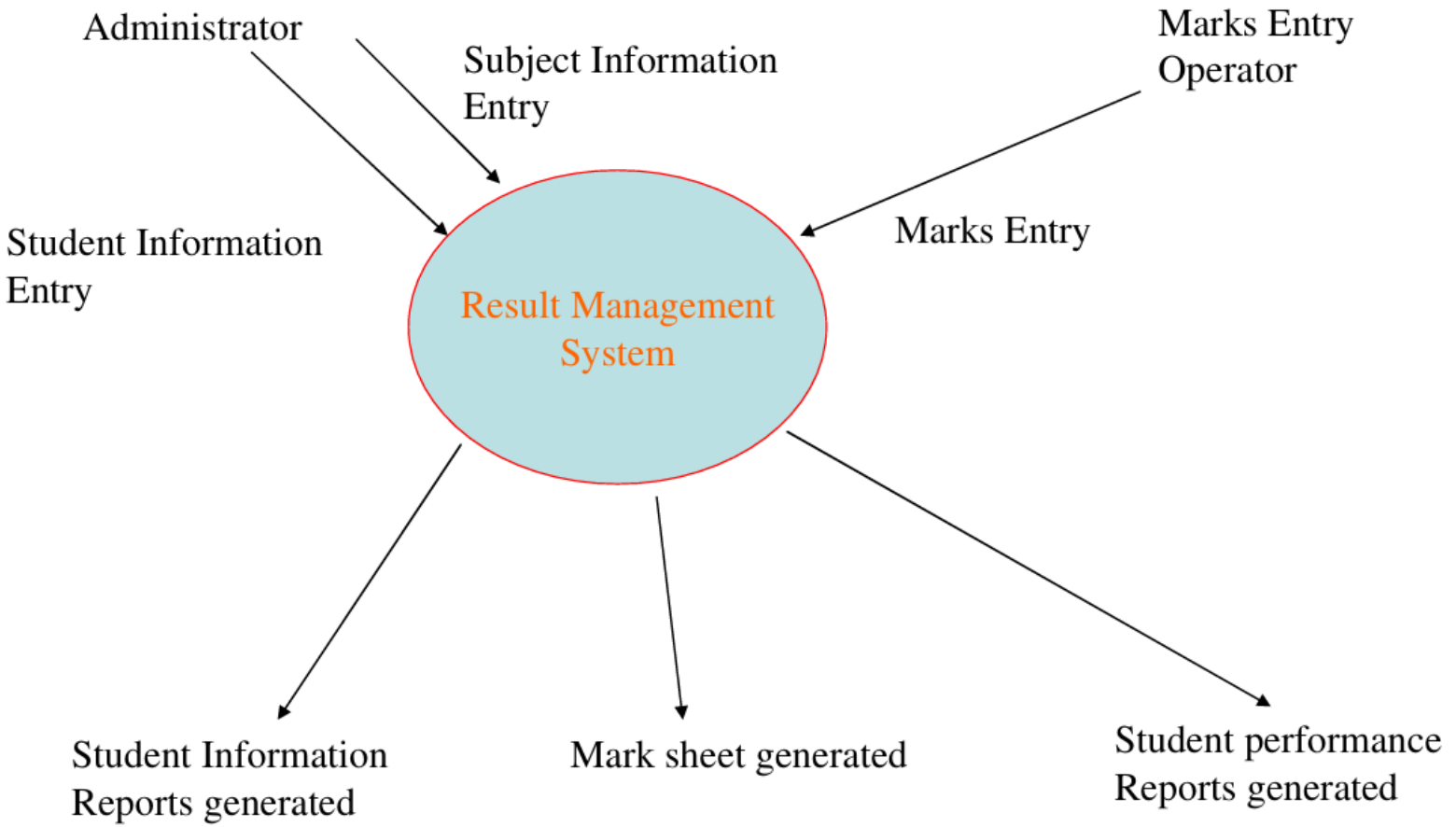
(1) Draw the context diagram. *.

The context diagram is a simple model that defines the "boundary and interface of the system with the external entities."



"The context diagram is also known as level-0 DFD (Data flow diagram)."

Requirements Analysis



Data Flow Diagrams

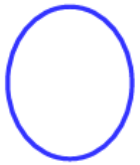
DFD show the flow of data through the system.

--All names should be unique

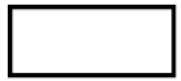
-- It is not a flow chart

-- Suppress logical decisions

-- Defer error conditions & handling until the end of the analysis

Symbol	Name	Function
	Data Flow	Connect process
	Process	Perform some transformation of its input data to yield output data.

Requirements Analysis

Symbol	Name	Function
	Source or sink	A source of system inputs or sink of system outputs
	Data Store	A repository of data, the arrowhead indicate net input and net outputs to store

Leveling

DFD represent a system or software at any level of abstraction.

A level 0 DFD is called fundamental system model or context model represents entire software element as a single bubble with input and output data indicating by incoming & outgoing arrows.

(Bubble charts)

(1) Data Flow Diagram. * (DFD)

The data flow diagram are used for modeling the requirement. DFD shows the flow of the data through the system. DFD is also know as data flow or bubble charts.

There are following components of DFD

(1) → arrow (data flow)

(2) ○ circle (process)

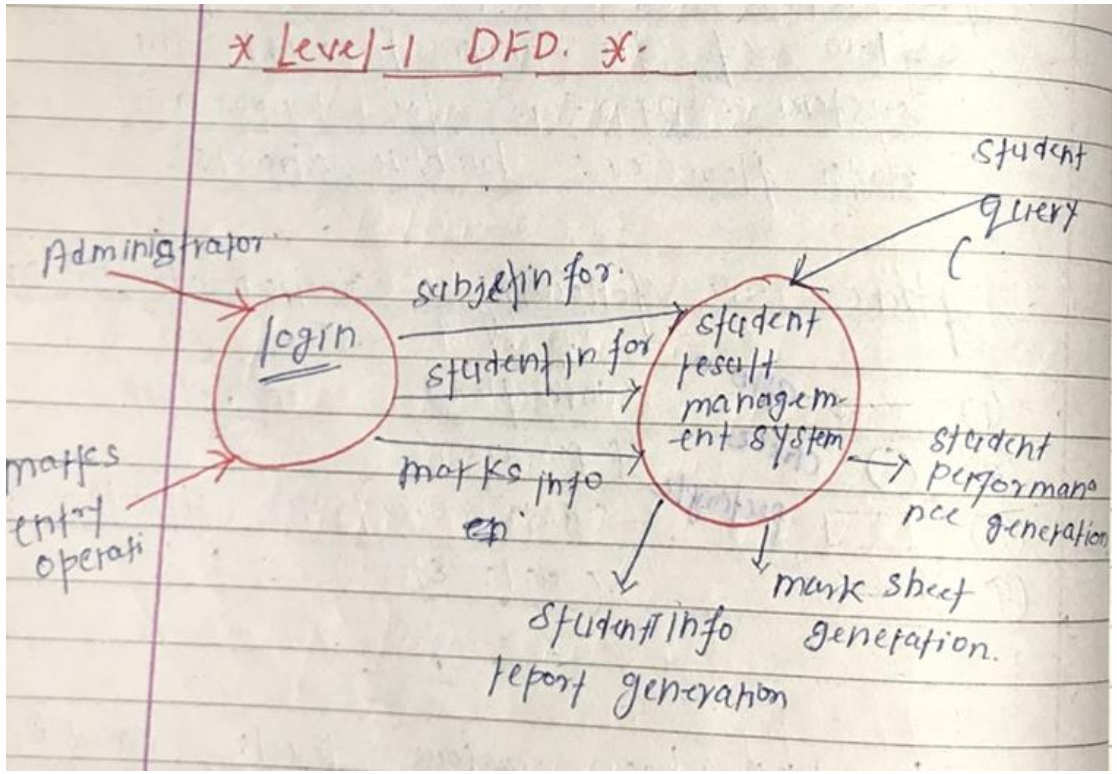
(3) ▭ rectangle (data source / I/P) (file to → another)

(4) ⇊ (data store)

The DFD has various levels based on the process and representation.

Level-0 DFD. *

Level-0 DFD is also known as context diagram. level-0 DFD has a single process which represent the boundries and interfaces from the externey entity that means the various input taken from the user or other data source and all the possible out put.



level one DFD has two level of process i.e. "the level-1 DFD is more descriptive to the level-0 DFD."

Each DFD must follow certain point-

① ⇒ "Every process within the DFD must have unique name."

② ⇒ The DFD is not a flow charts i.e. DFD represents the flow of the data instead of flow of events. i.e. "DFD does not use any component for decision making, loop construct."

2 * E-R Diagram *

E-R diagram is a another tool for requirement analysis. It provides a detail logical representation of the data for the system. It This diagram contain three main component.

① Entity ② Attribute ③ Relationship.

(1) Entity *

Entity is name of any thing.

ex. program, student.

(Nouns are always an entity).

Ex - E-R. diagram entity are represented by rectangle.

An entity may have sub entity connected through the relationship.

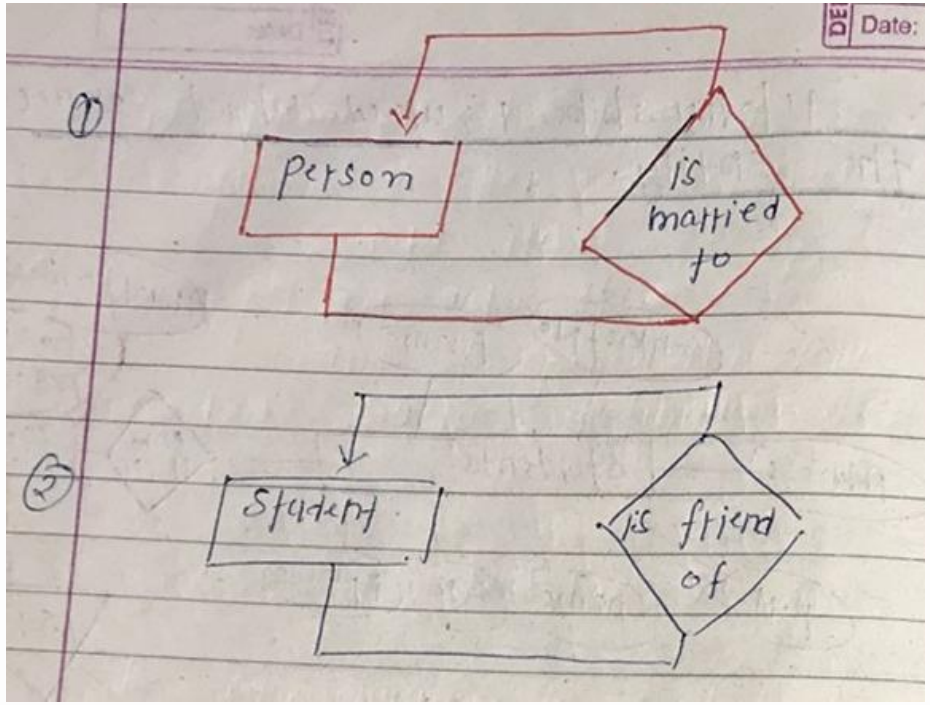
ex. regular prog, Distance education prog is a sub-entity of an entity program.

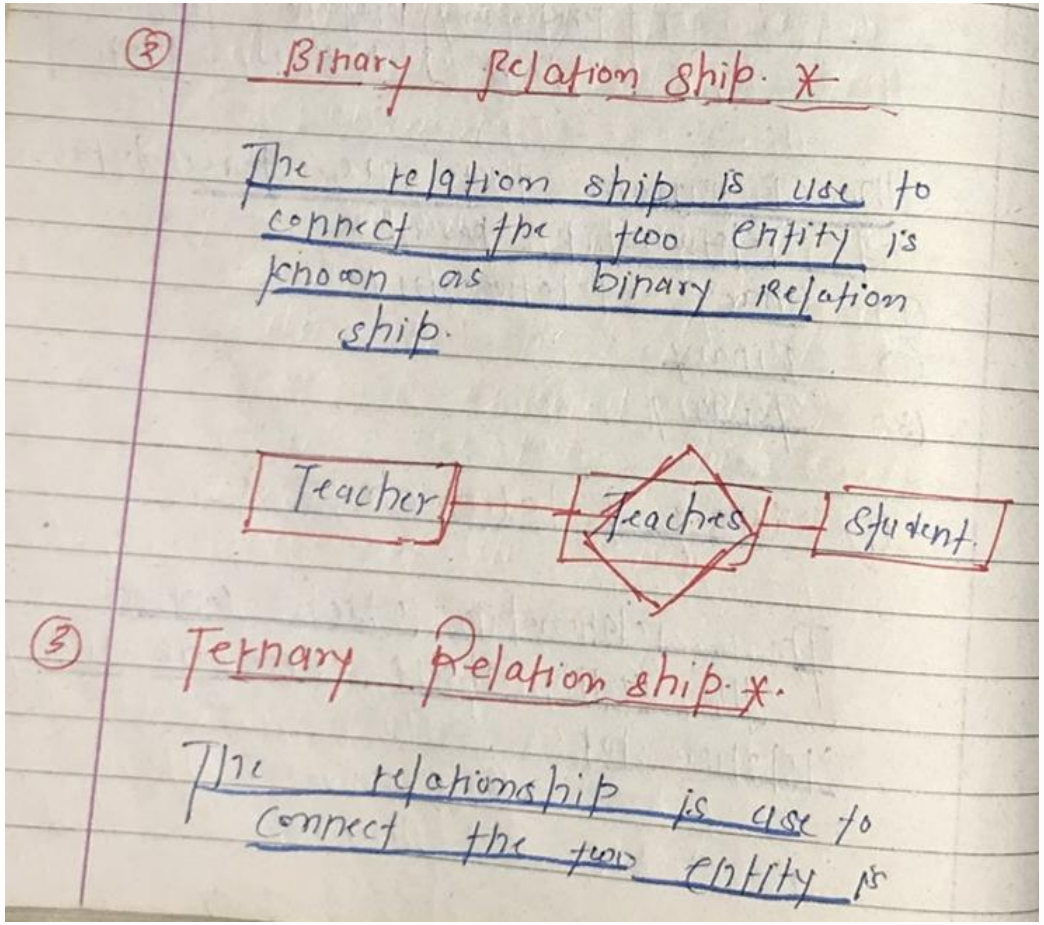
In ER diagram there are three types of relationship possible.

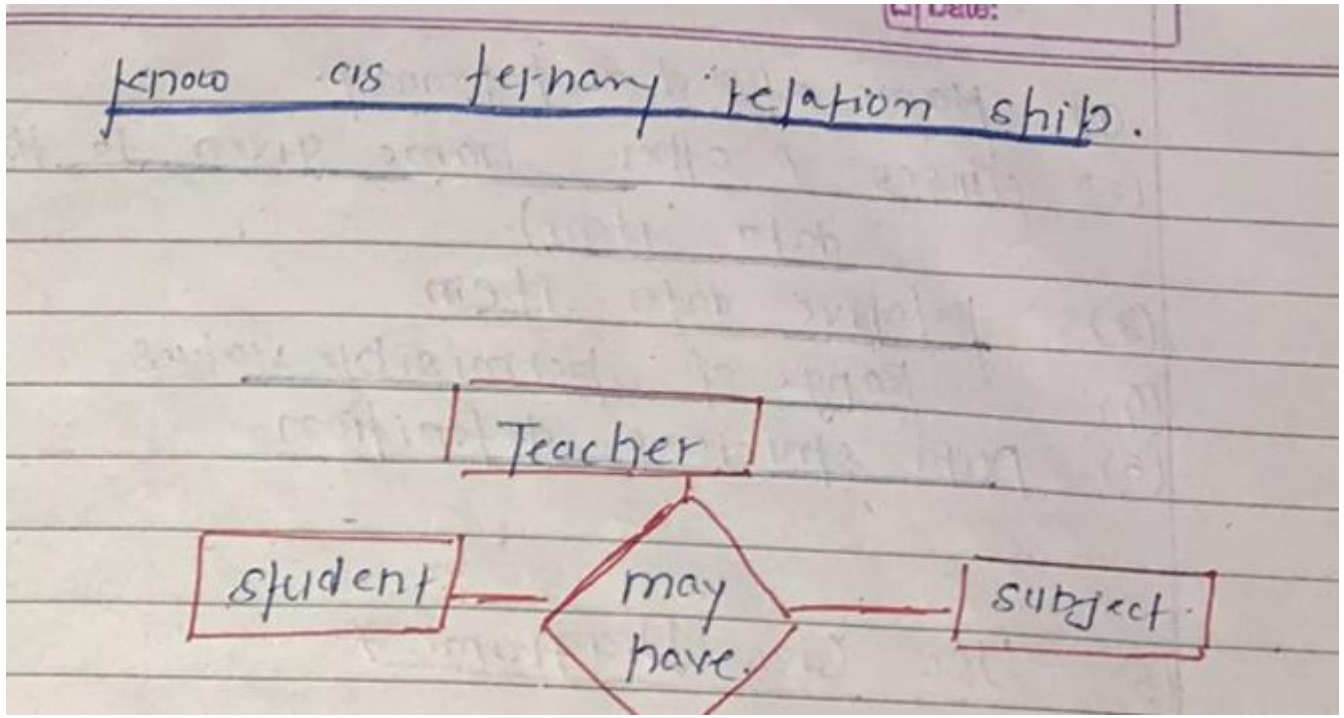
(1) Unary relationship
(2) Binary ,,
(3) Ternary ,,

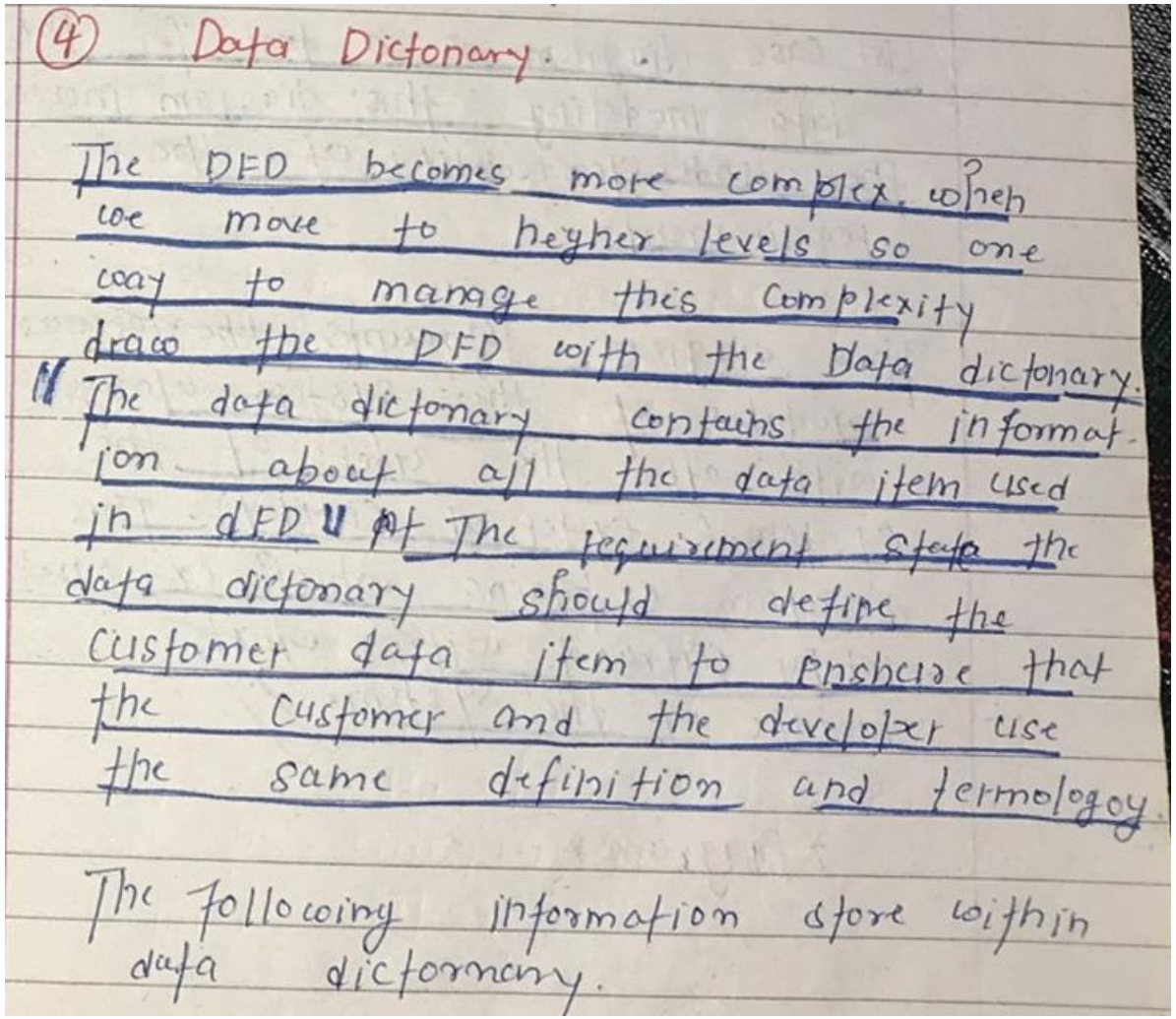
1) Unary Relationship. *

The relationship which connect the entity itself know as unray relation ship.




2 Binary Relation ship. *
The relation ship is use to connect the two entity is known as binary relation ship.
Teacher
Teaches
Student.
3 Ternary Relation ship. *.
The relationship is use to connect the two entity is

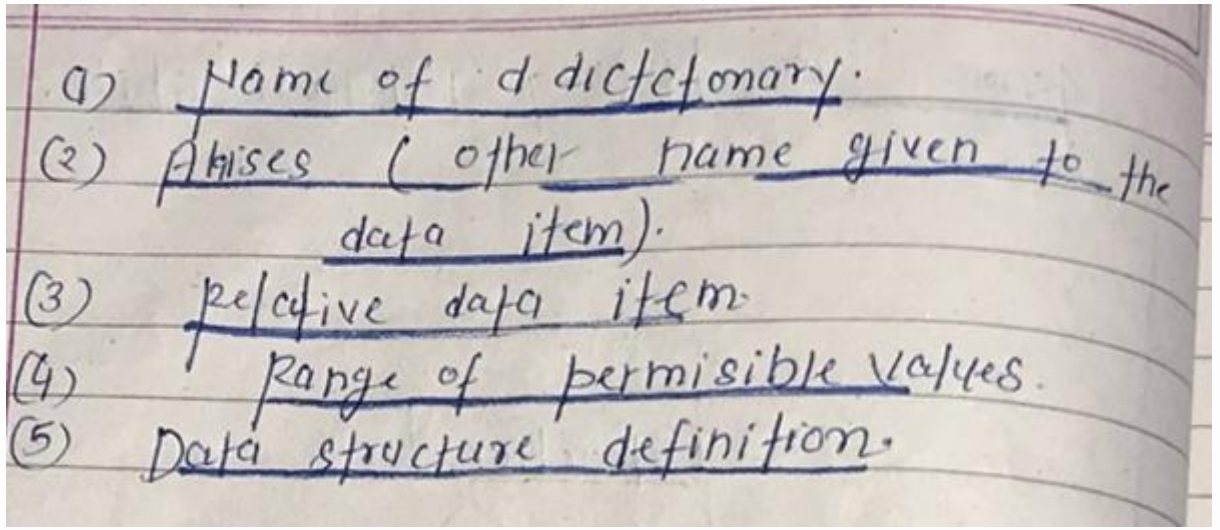

known as ternary relation ship.
Teacher
student
may have.
subject.



④ Data Dictonary.

The DFD becomes more complex. when we move to heyher levels so one way to manage this complexity draw the DFD with the Data dictonary. The data dictonary contains the information about all the data item used in DFD At The requirement state the data dictonary should define the customer data item to ensheure that the customer and the developer use the same definition and termology.

The following information store within data dictornary.



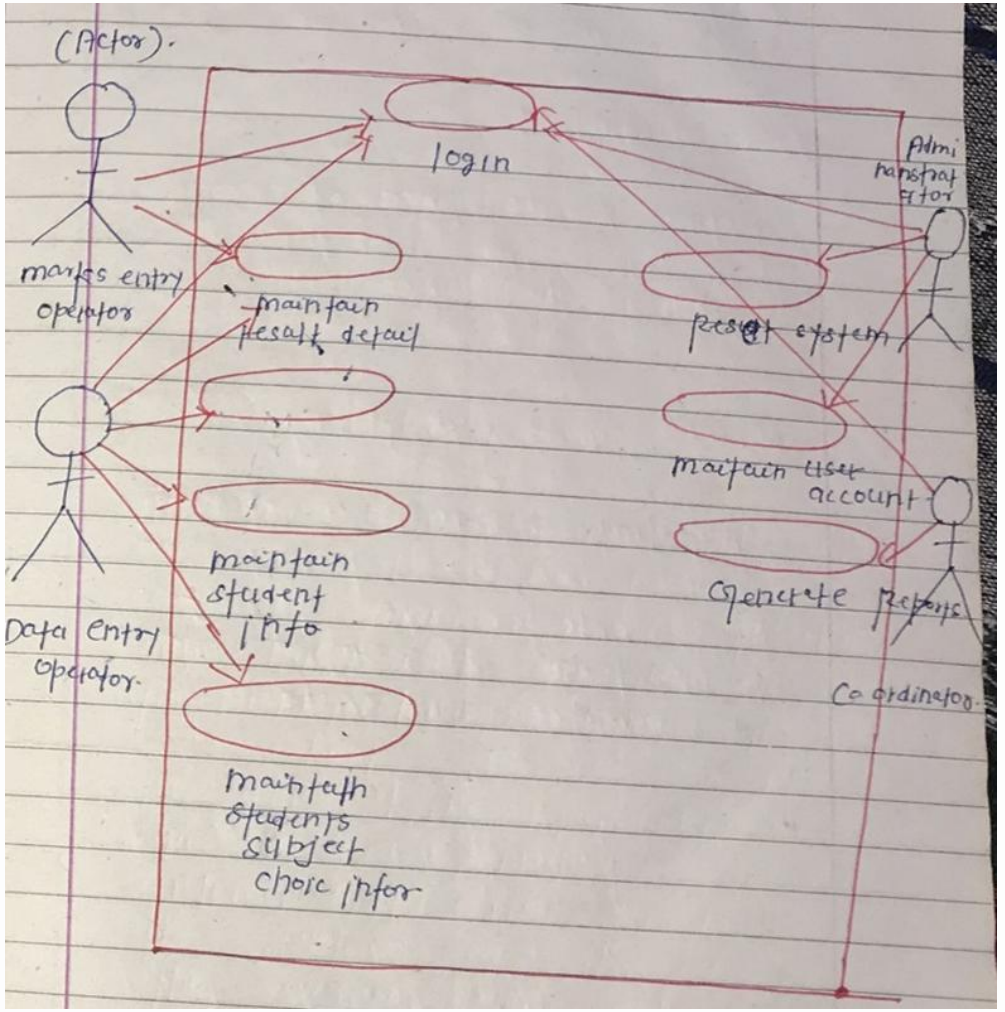
(1) Name of d dictctonary.
(2) Aliases (other name given to the data item).
(3) Relative data item.
(4) Range of permisible values.
(5) Data structure definition.

(5) Use Case Diagram. *

Use case diagram is a tool for the data modeling this diagram increase the understandability of the requirement.

This diagram represents the various modules of the system along with all the user of the system (external entity). This diagram explains which external entity interact with which module of the system.

* Diagram *



Data Dictionaries

DFD ⟶ DD

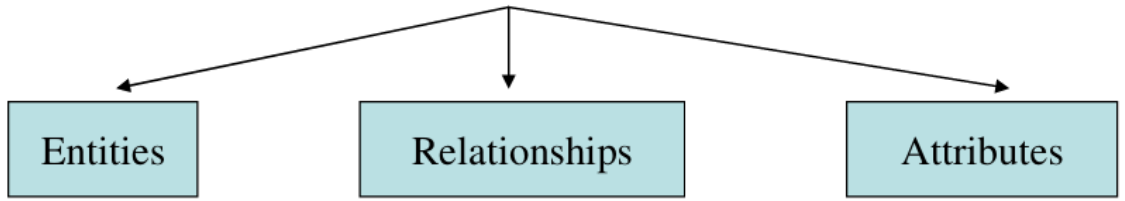
Data Dictionaries are simply repositories to store information about all data items defined in DFD.

Includes :

- Name of data item
- Aliases (other names for items)
- Description/Purpose
- Related data items
- Range of values
- Data flows
- Data structure definition

Entity-Relationship Diagrams

It is a detailed logical representation of data for an organization and uses three main constructs.



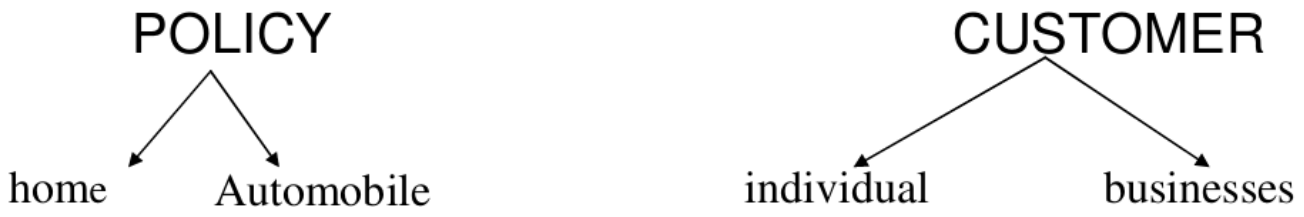
Entities

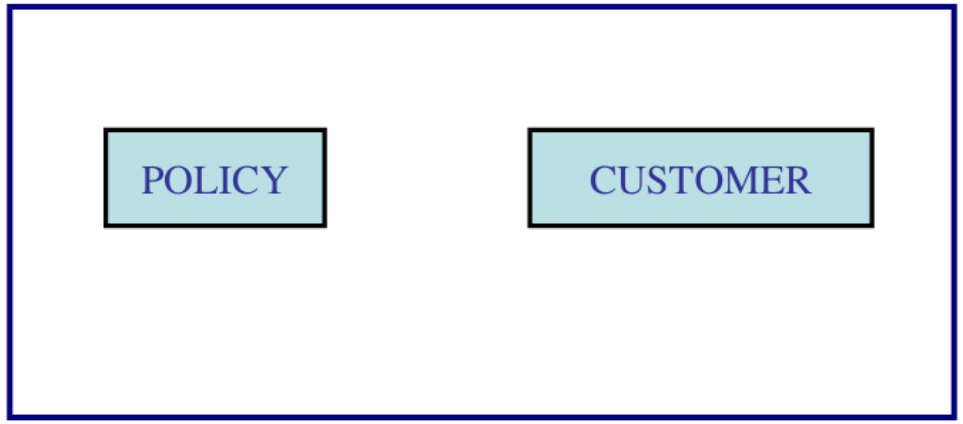
Fundamental thing about which data may be maintained. Each entity has its own identity.

Entity Type is the description of all entities to which a common definition and common relationships and attributes apply.

Entity-Relationship Diagrams

Consider an insurance company that offers both home and automobile insurance policies .These policies are offered to individuals and businesses.





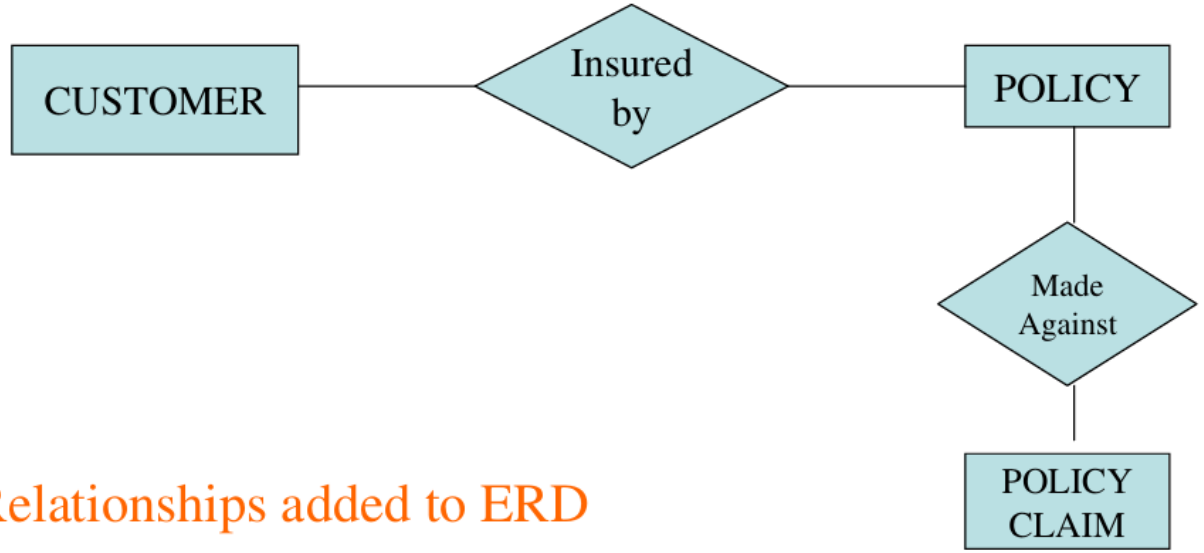
Relationships

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.

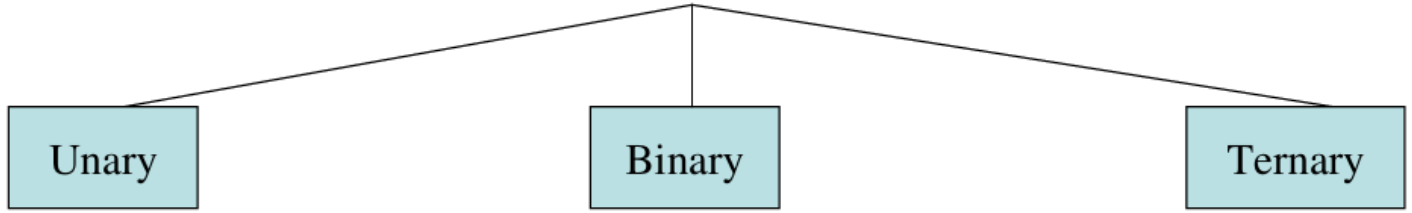


Relationships added to ERD

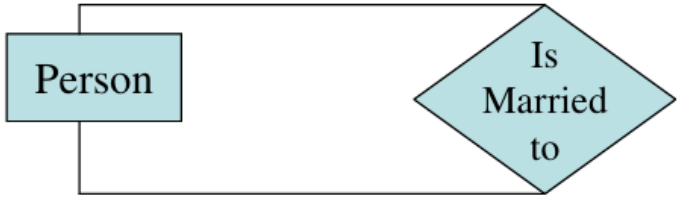
Entity-Relationship Diagrams

Degree of relationship

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



Unary relationship

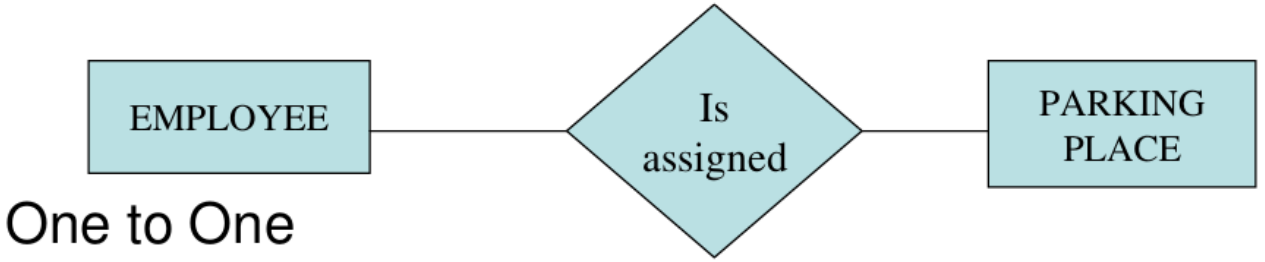


One to One

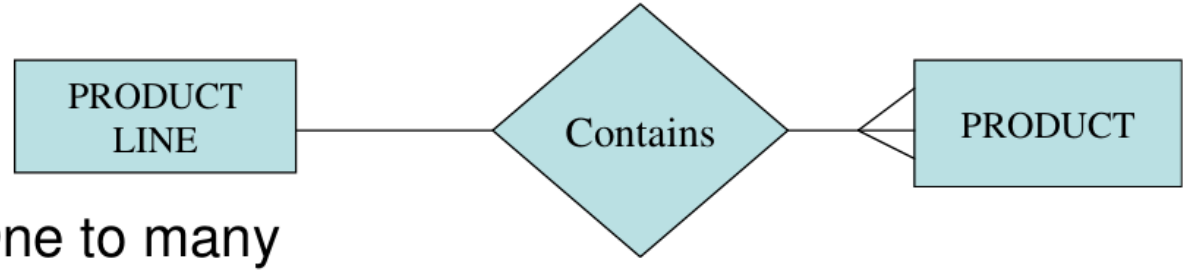
One to many

Entity-Relationship Diagrams

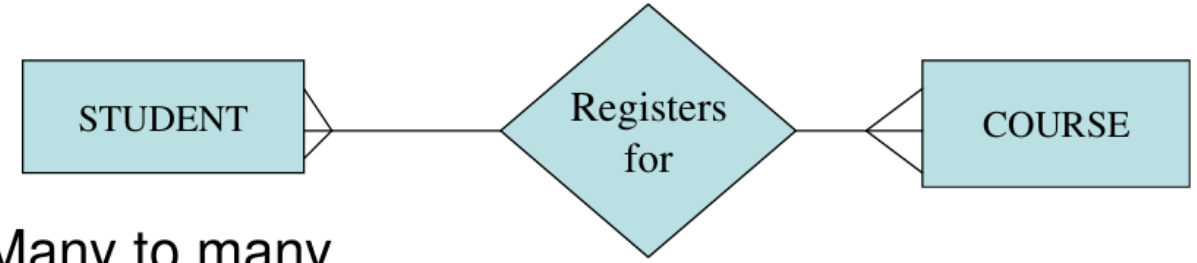
Binary Relationship



One to One



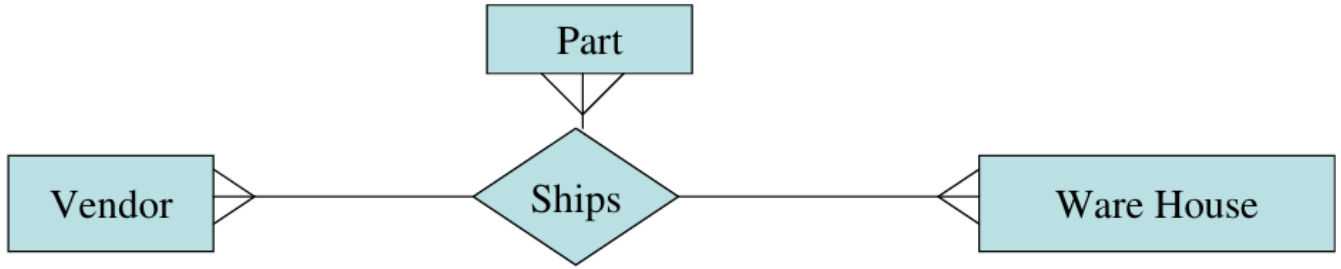
One to many



Many to many

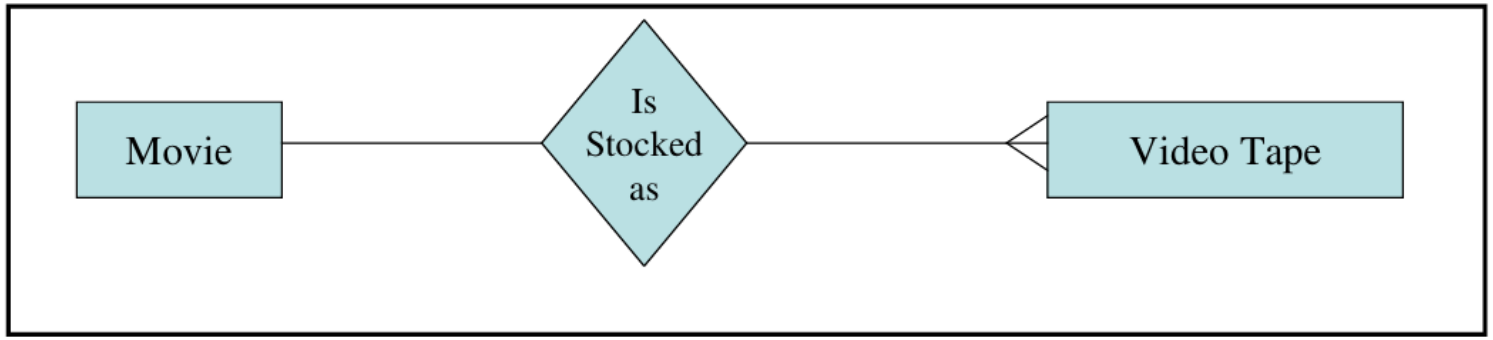
Entity-Relationship Diagrams

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

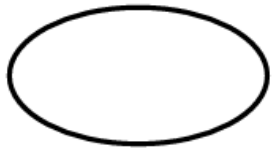


Entity-Relationship Diagrams

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

Entity-Relationship Diagrams

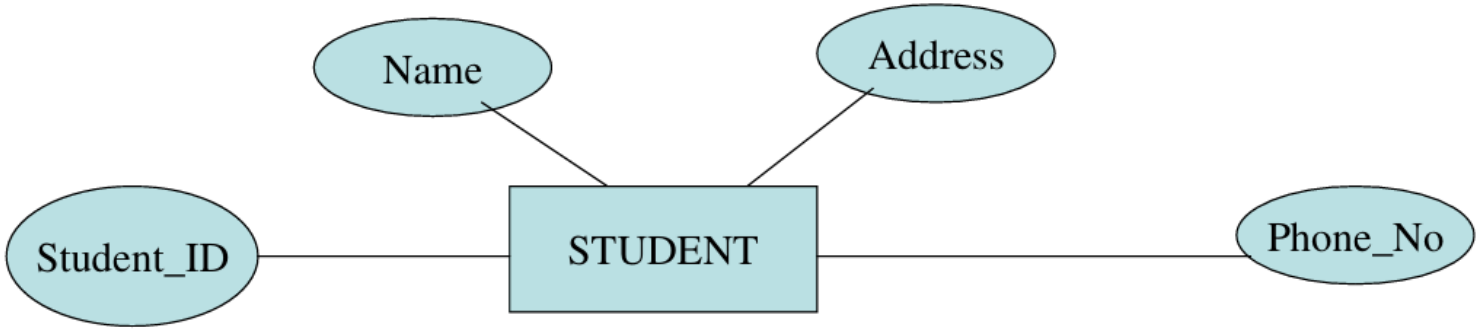
A candidate key is an attribute or combination of attributes that uniquely identifies each instance of an entity type.

Student_ID ⟶ Candidate Key

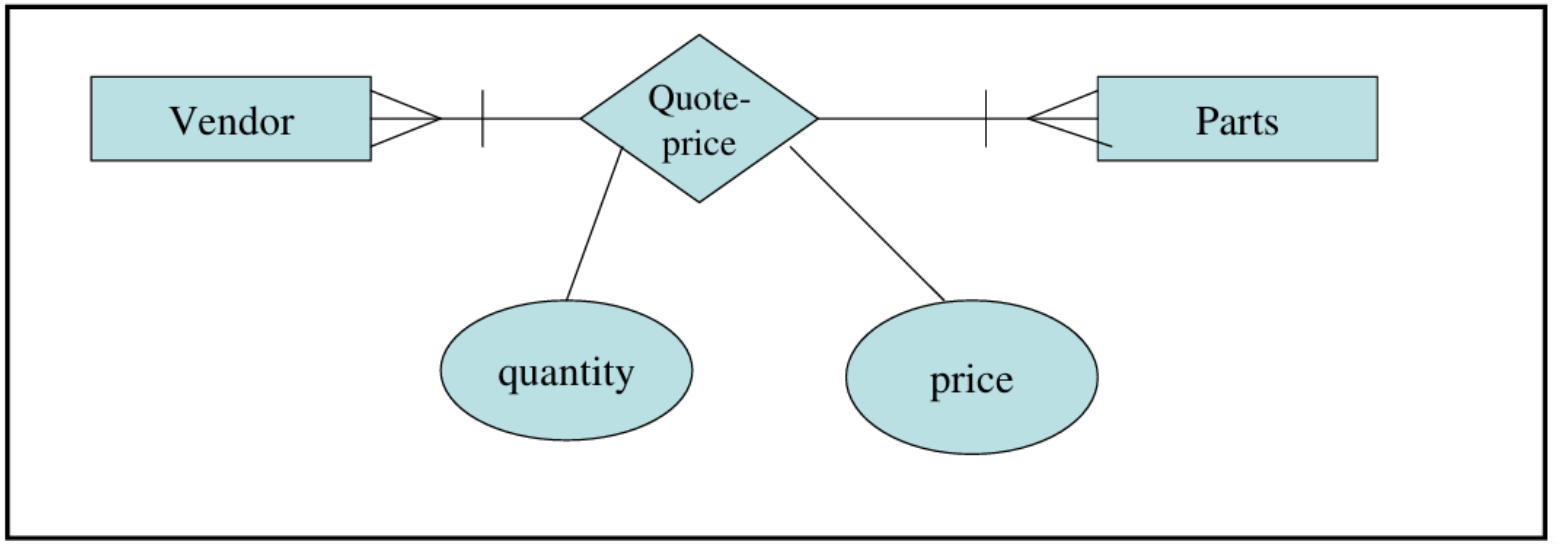
If there are more candidate keys, one of the key may be chosen as the Identifier.
It is used as unique characteristic for an entity type.

Identifier

Entity-Relationship Diagrams



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



Nature of SRS

Basic Issues

- Functionality
- External Interfaces
- Performance
- Attributes
- Design constraints imposed on an Implementation

Requirements Documentation

SRS Should

-- Correctly define all requirements

-- not describe any design details

-- not impose any additional constraints

Characteristics of a good SRS

An SRS Should be

✓ Correct

✓ Unambiguous

✓ Complete

✓ Consistent

Requirements Documentation

- ✓ Ranked for important and/or stability
- ✓ Verifiable
- ✓ Modifiable
- ✓ Traceable

Requirements Documentation

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.

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.

Requirements Documentation

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.

Organization of the SRS

IEEE has published guidelines and standards to organize an SRS.
First two sections are same. The specific tailoring occurs in section-3.

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

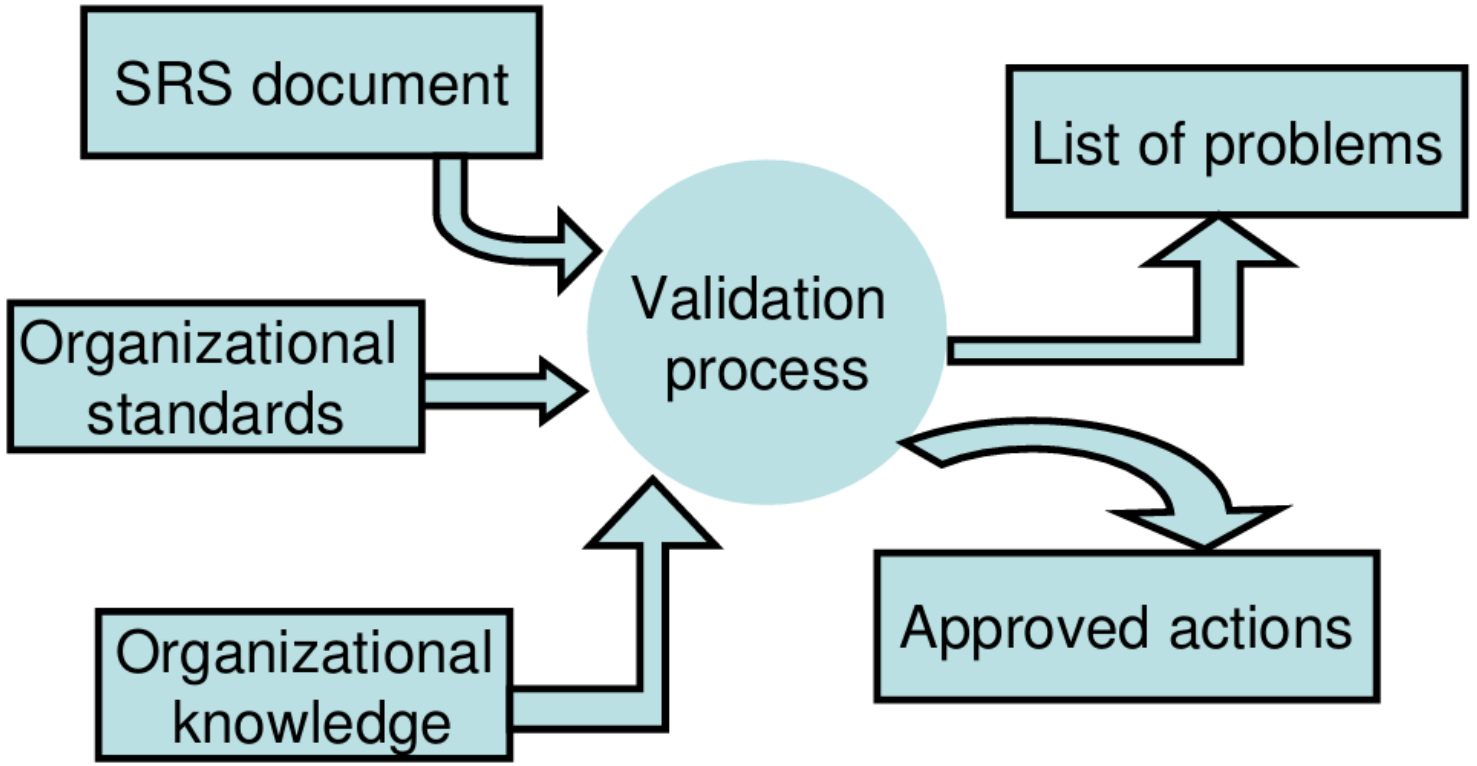
Requirements Documentation

2. The Overall Description

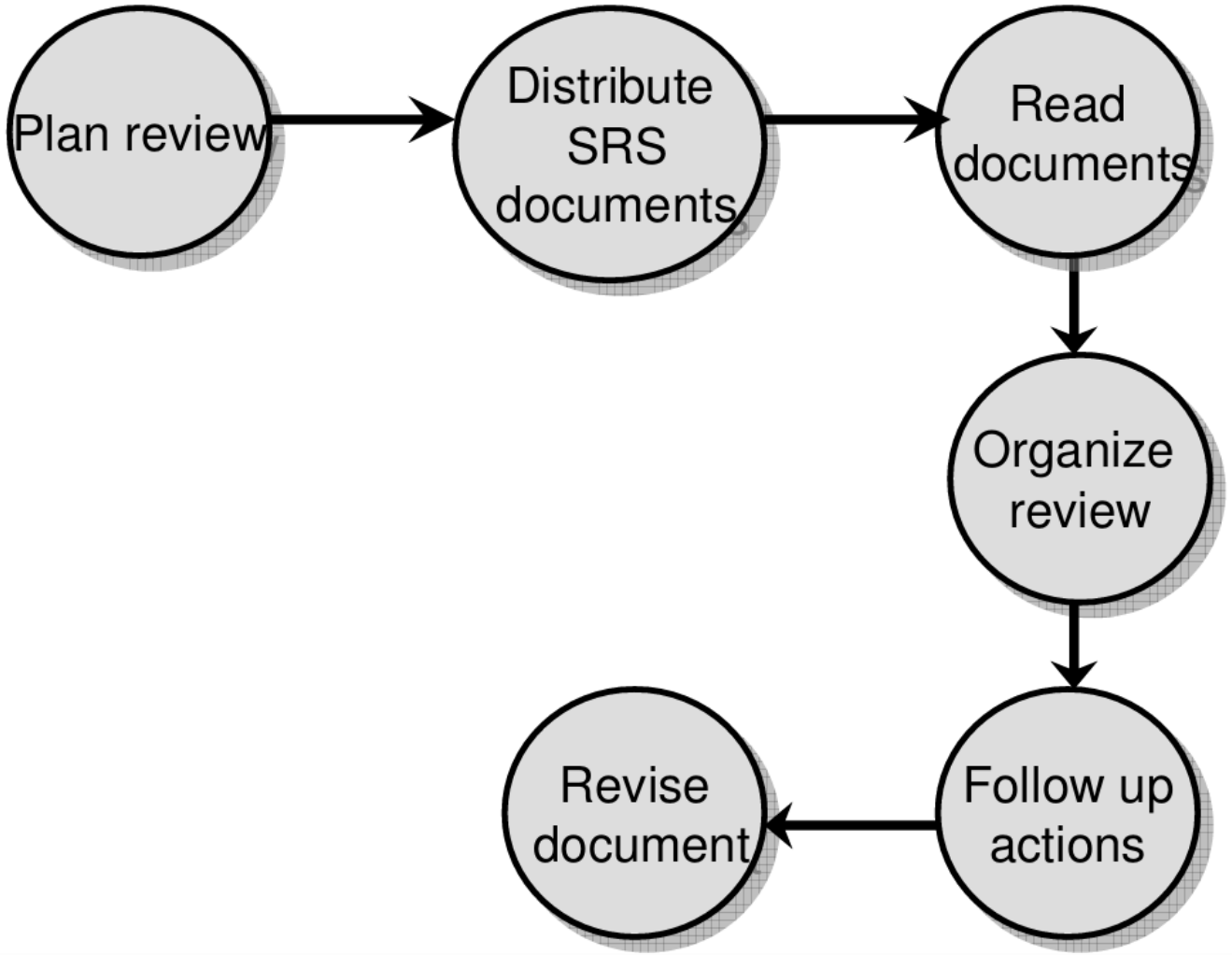
2.1 Product Perspective

- 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

Requirements Validation



Requirements Review Process



3.19. An airline reservation is an association between a passenger, a flight, and a seat. Select a few

pertinent attributes for each of these entity types and represent a reservation in an E–R diagram.

3.20. A department of computer science has usual resources and usual users for these resources. A software is to be developed so that resources are assigned without conflict. Draw a DFD specifying the above system.

3.21. Draw a DFD for result preparation automation system of B. Tech. courses (or MCA program) of any university. Clearly describe the working of the system. Also mention all assumptions made by you.

3.22. Write short notes on

(*i*) Data flow diagram

(*ii*) Data dictionary.

3.23. Draw a DFD for borrowing a book in a library which is explained below: "A borrower can borrow a book if it is available else he/she can reserve for the book if he/she so wishes. He/she can borrow a maximum of three books".

3.24. Draw the E–R diagram for a hotel reception desk management.

Explain why, for large software systems development, is it recommended that prototypes should be "throw-away" prototype ?

Object Oriented Metrics

Terminologies

S.No	Term	Meaning/purpose
1	Object	Object is an entity able to save a state (information) and offers a number of operations (behavior) to either examine or affect this state.
2	Message	A request that an object makes of another object to perform an operation.
3	Class	A set of objects that share a common structure and common behavior manifested by a set of methods; the set serves as a template from which object can be created.
4	Method	an operation upon an object, defined as part of the declaration of a class.
5	Attribute	Defines the structural properties of a class and unique within a class.
6	Operation	An action performed by or on an object, available to all instances of class, need not be unique.

Object Oriented Metrics

Terminologies

S.No	Term	Meaning/purpose
7	Instantiation	The process of creating an instance of the object and binding or adding the specific data.
8	Inheritance	A relationship among classes, where in an object in a class acquires characteristics from one or more other classes.
9	Cohesion	The degree to which the methods within a class are related to one another.
10	Coupling	Object A is coupled to object B, if and only if A sends a message to B.

- Measuring on class level
  - coupling
  - inheritance
  - methods
  - attributes
  - cohesion
- Measuring on system level

Object Oriented Metrics

Coupling Metrics:

- Response for a Class (RFC)
  - Number of methods (internal and external) in a class.

- Data Abstraction Coupling(DAC)
  - Number of Abstract Data Types in a class.

- Coupling between Objects (CBO)
  - Number of other classes to which it is coupled.

Object Oriented Metrics

- Message Passing Coupling (MPC)
  - Number of send statements defined in a class.
- Coupling Factor (CF)
  - Ratio of actual number of coupling in the system to the max. possible coupling.

Software Design

- More creative than analysis
- Problem solving activity

WHAT IS DESIGN

'HOW'

Software design document (SDD)

Software Design

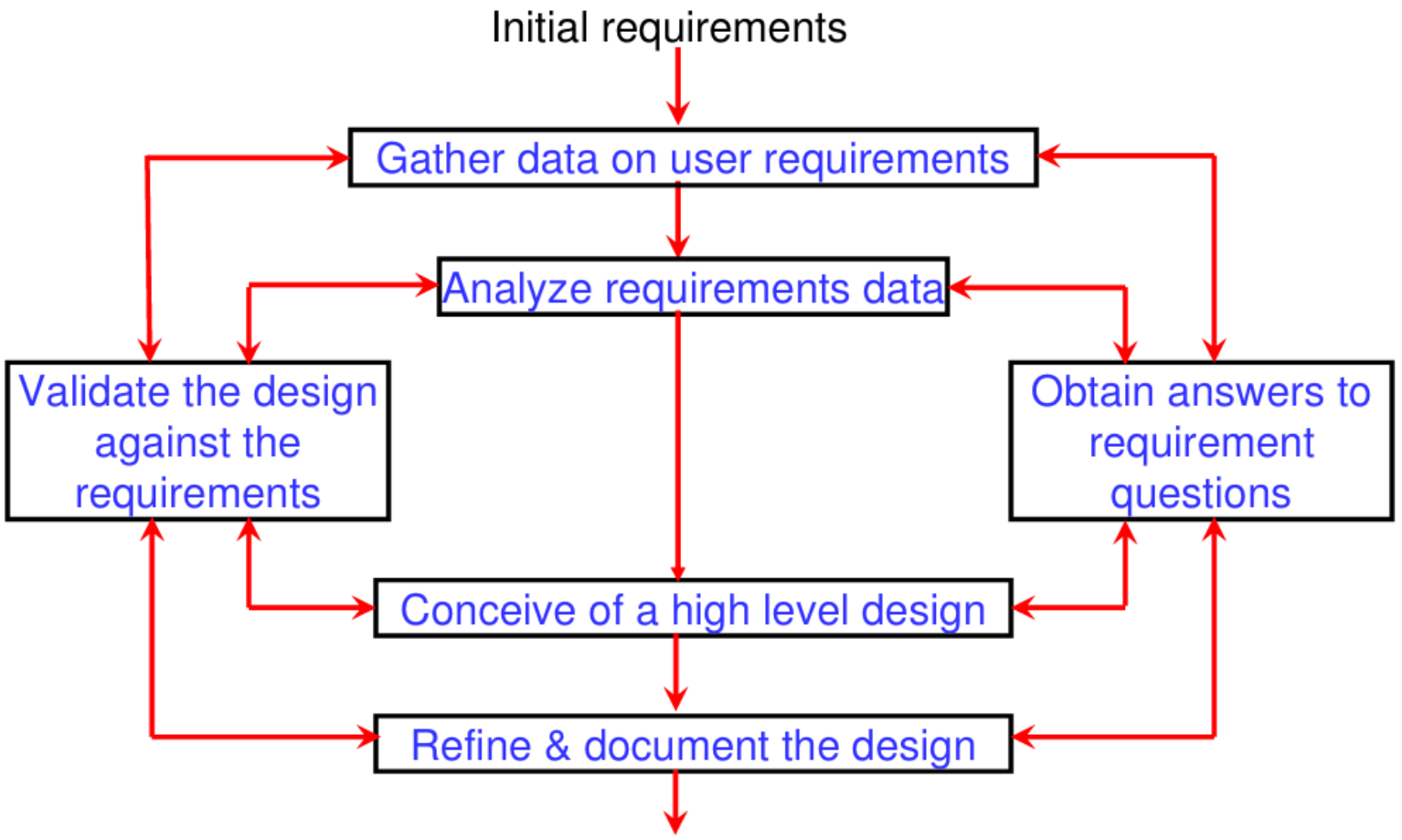
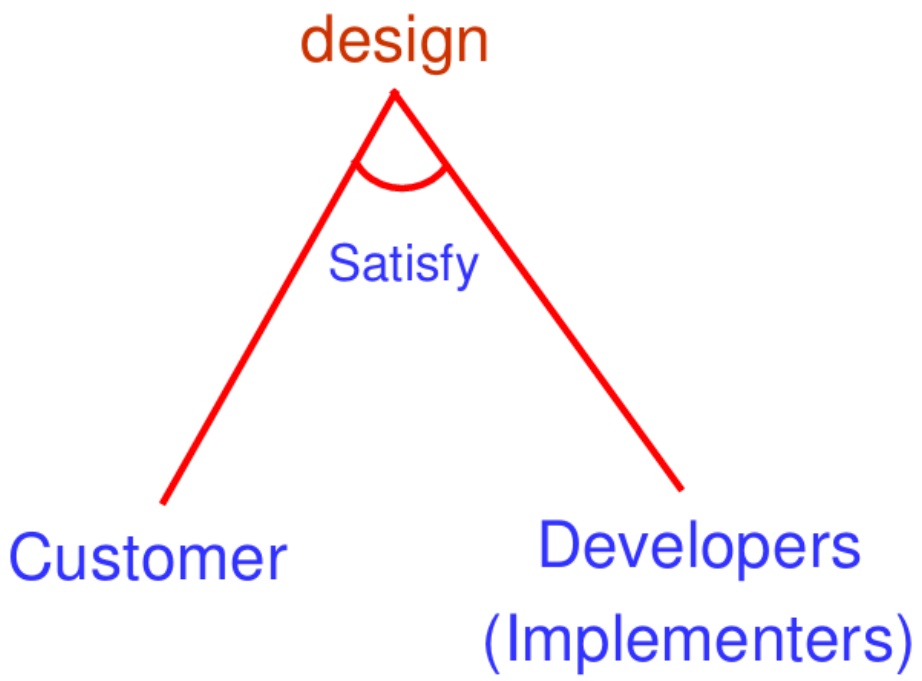


Fig. 1 : Design framework

Software Design



Conceptual Design and Technical Design

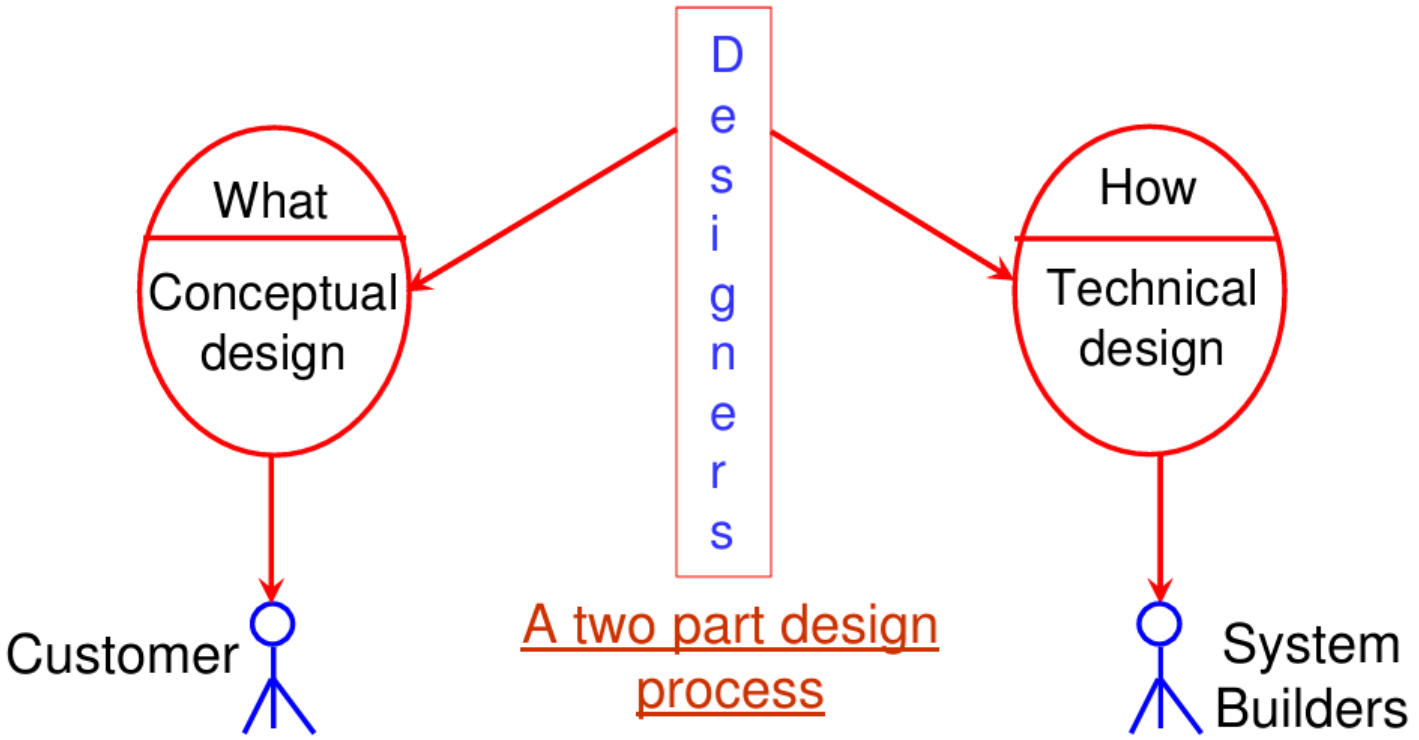


Fig. 2 : A two part design process

Software Design

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?

Software Design

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.

Software Design

The design needs to be

- Correct & complete
- Understandable
- At the right level
- Maintainable

MODULARITY

There are many definitions of the term module. Range is from :

i. Fortran subroutine

ii. Ada package

iii. Procedures & functions of PASCAL & C

iv. C++ / Java classes

v. Java packages

vi. Work assignment for an individual programmer

Software Design

All these definitions are correct. A modular system consist of well defined manageable units with well defined interfaces among the units.

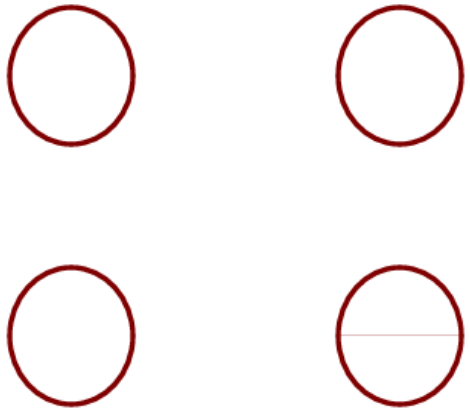
Software Design

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 software product.

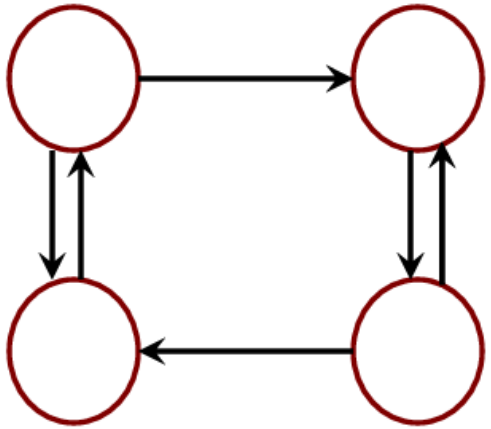
Module Coupling

Coupling is the measure of the degree of interdependence between modules.



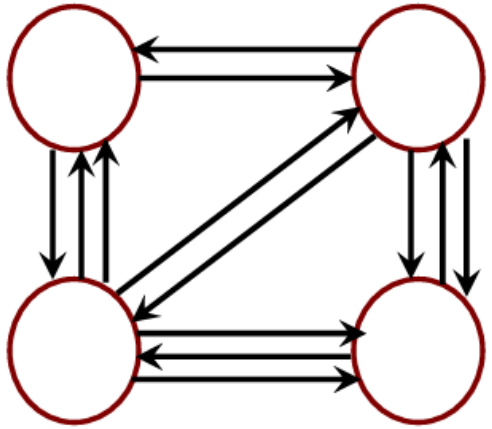
(Uncoupled : no dependencies)

(a)



Loosely coupled:
some dependencies

(B)



Highly coupled:
many dependencies

(C)

Fig. 5 : Module coupling

Software Design

This can be achieved as:

- 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.

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

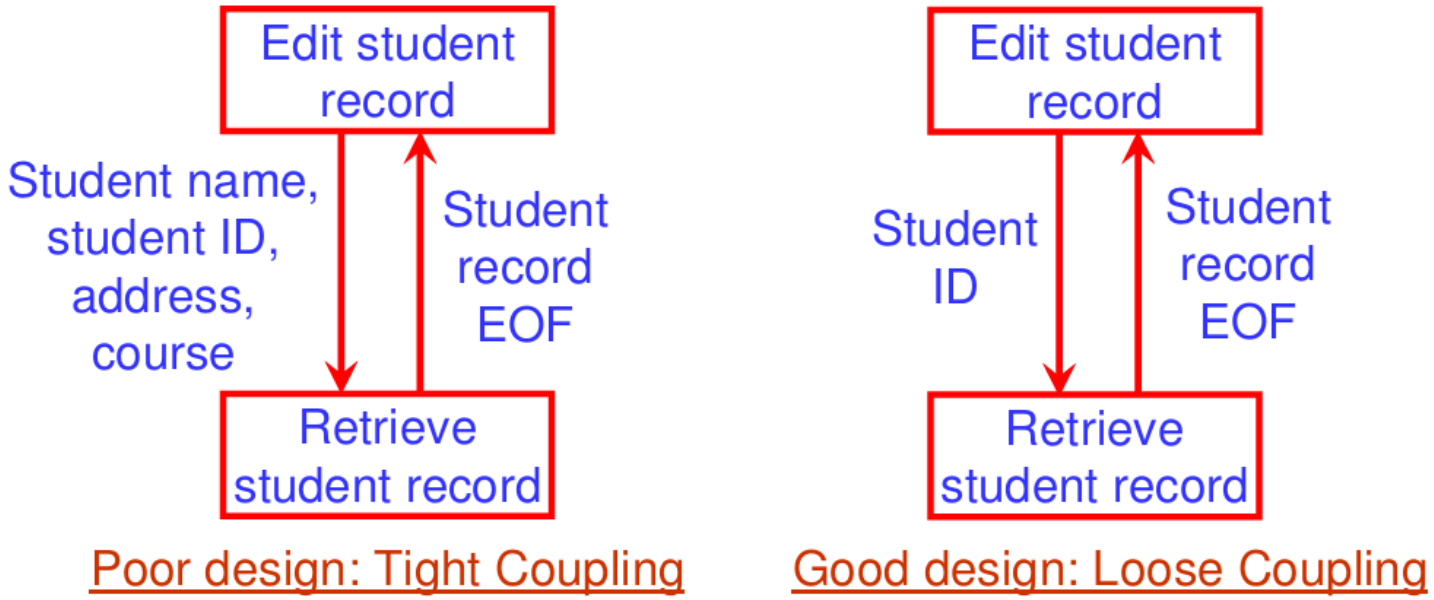


Fig. 6 : Example of coupling

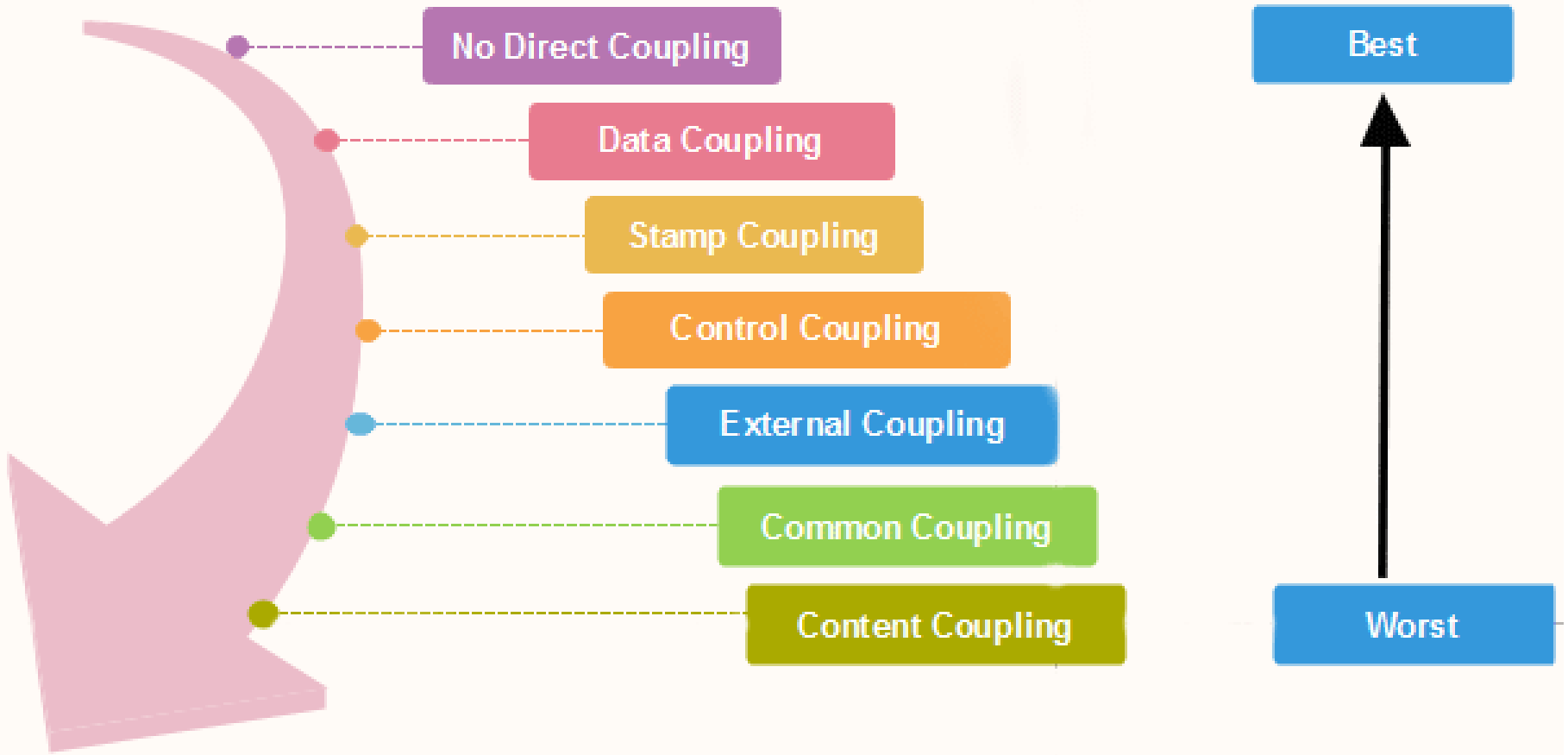
Data coupling	Best
Stamp coupling	
Control coupling	
External coupling	
Common coupling	
Content coupling	Worst

Fig. 7 : The types of module coupling

Given two procedures A & B, we can identify number of ways in which they can be coupled.

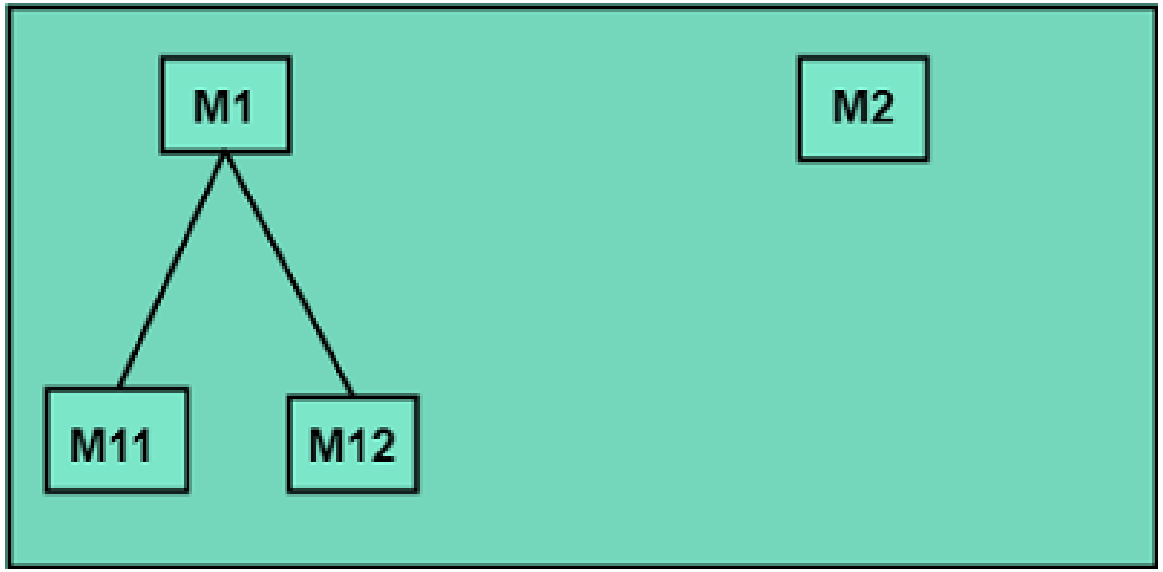
Types of Modules Coupling

There are various types of module Coupling are as follows:



No Direct Coupling:

There is no direct coupling between M1 and M2.

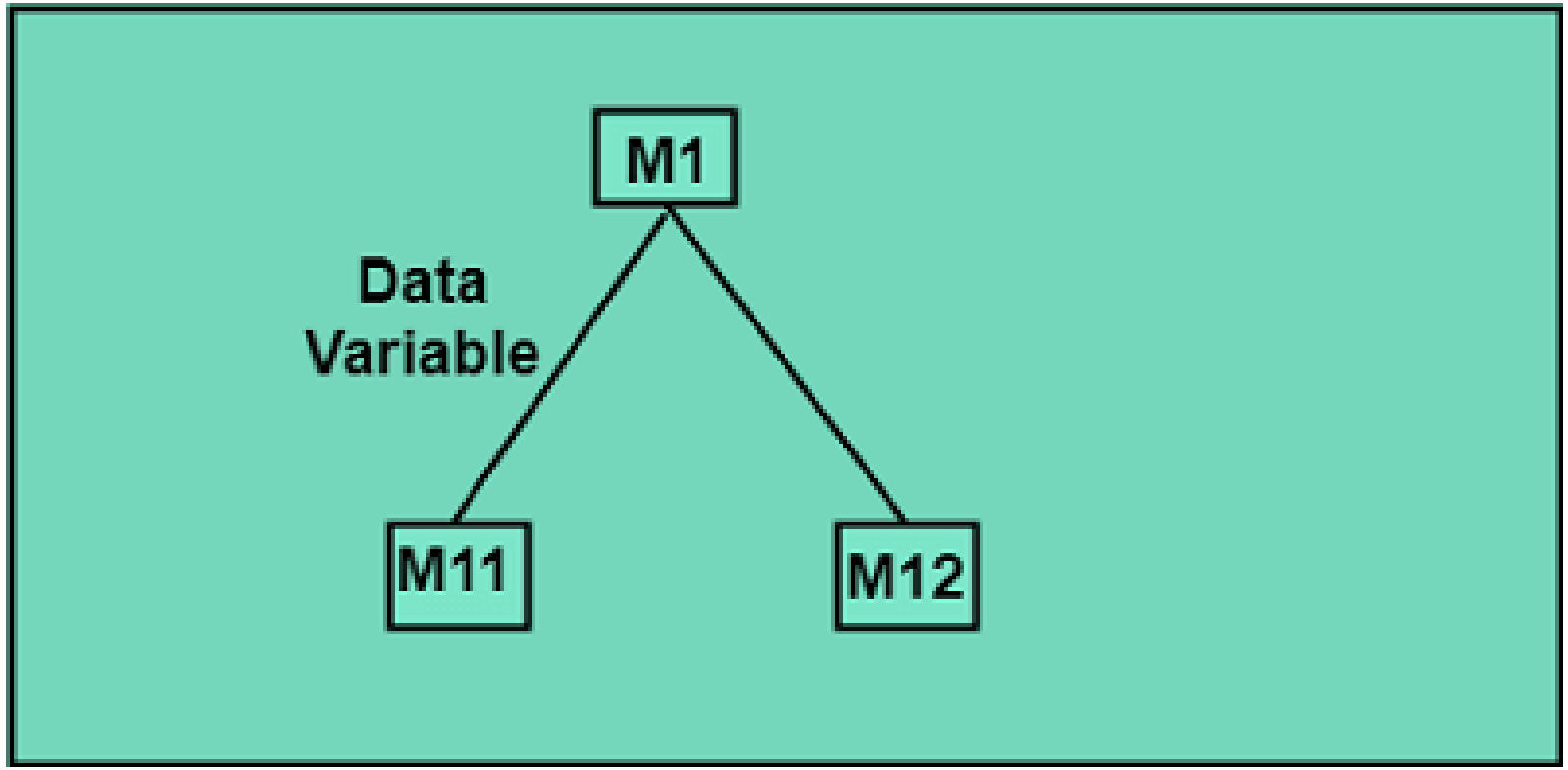


Data Coupling:

- When data of one module is passed to another module, this is called data coupling.

Data coupling

The dependency between module A and B is said to be data coupled if their dependency is based on the fact they communicate by only passing of data. Other than communicating through data, the two modules are independent.

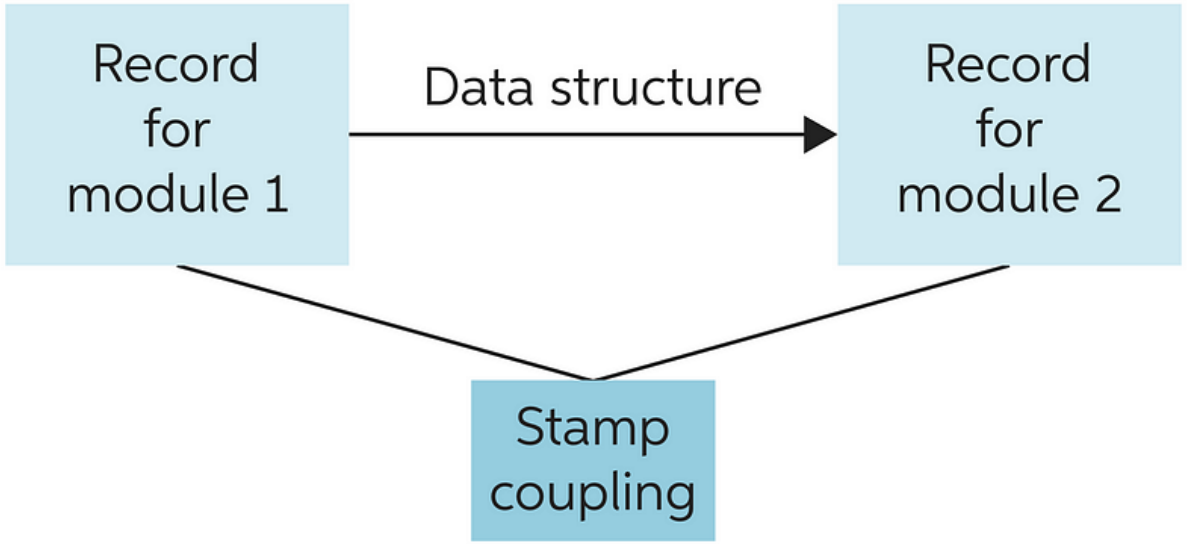

M1
Data
Variable
M11
M12

Stamp Coupling:

- Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc.
- When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled.
- For example, passing structure variable in C or object in C++ language to a module.

Stamp coupling

Stamp coupling occurs between module A and B when complete data structure is passed from one module to another.



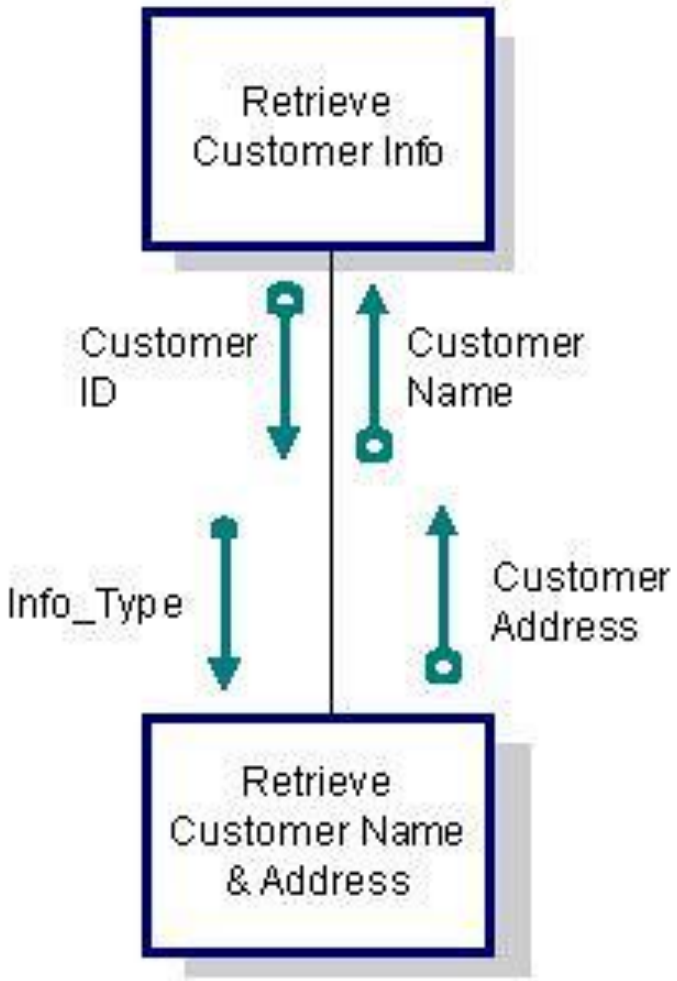
Control Coupling:

- **Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.**

Control coupling

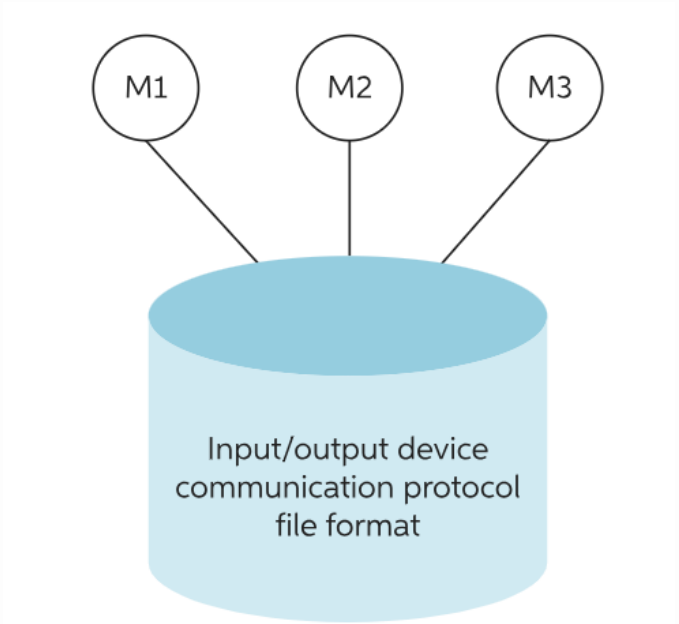
Module A and B are said to be control coupled if they communicate by passing of control information. This is usually accomplished by means of flags that are set by one module and reacted upon by the dependent module.

CONTROL COUPLE



External Coupling:

- External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface.
- This is related to communication to external tools and devices.



Common Coupling:

- Two modules are common coupled if they share information through some global data items.

Common coupling

With common coupling, module A and module B have shared data. Global data areas are commonly found in programming languages. Making a change to the common data means tracing back to all the modules which access that data to evaluate the effect of changes.

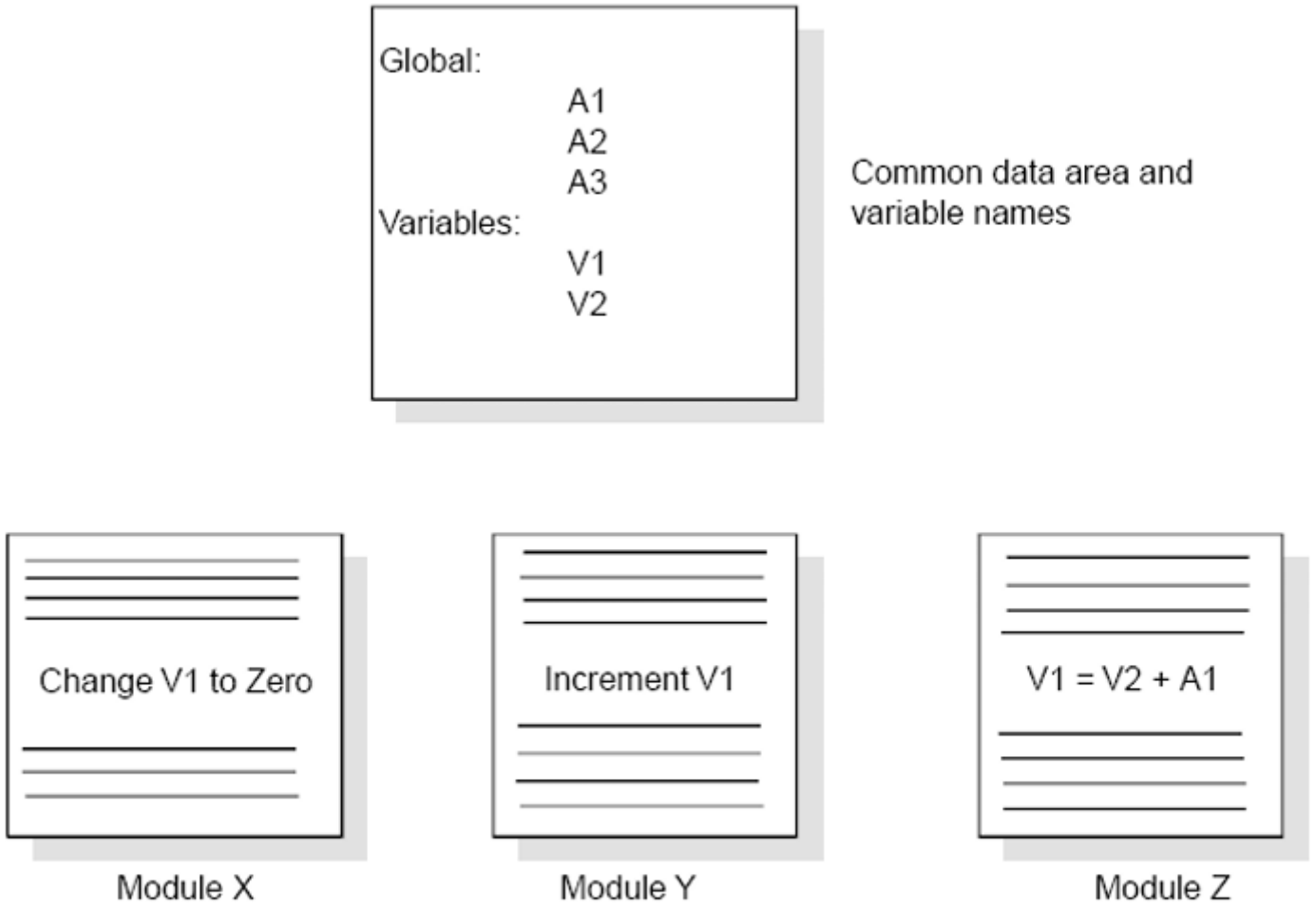


Fig. 8 : Example of common coupling

Content Coupling:

- Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

Content coupling

Content coupling occurs when module A changes data of module B or when control is passed from one module to the middle of another. In Fig. 9, module B branches into D, even though D is supposed to be under the control of C.

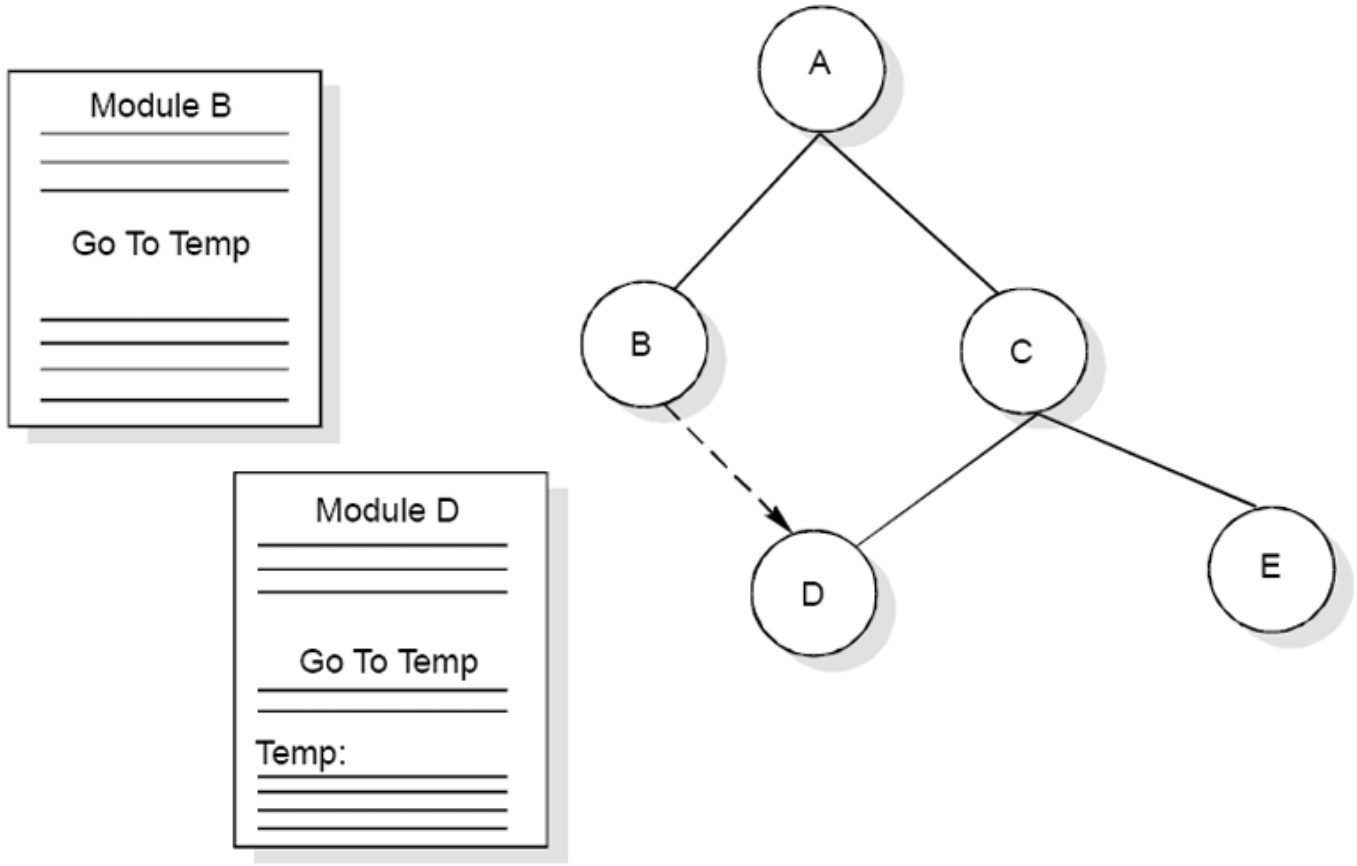


Fig. 9 : Example of content coupling

Module Cohesion

Cohesion is a measure of the degree to which the elements of a module are functionally related.

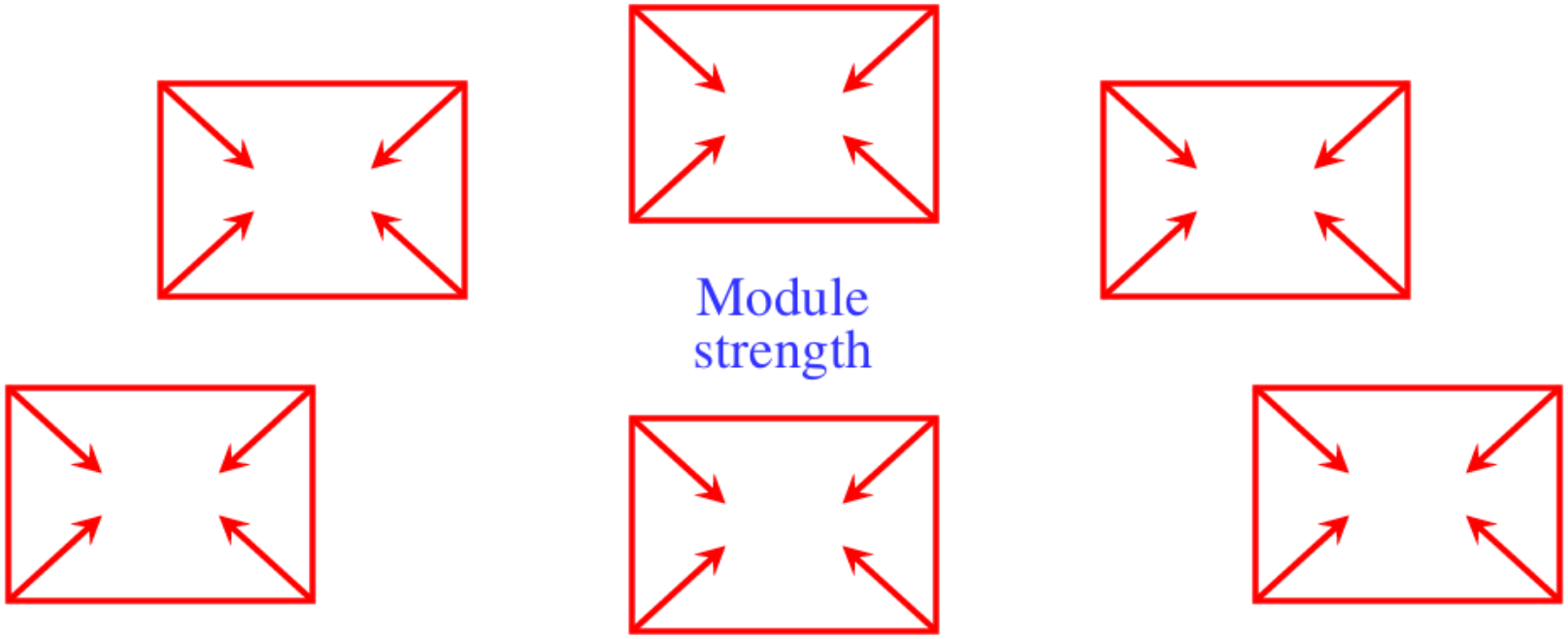


Fig. 10 : Cohesion=Strength of relations within modules

Types of cohesion

- Functional cohesion
- Sequential cohesion
- Procedural cohesion
- Temporal cohesion
- Logical cohesion
- Coincident cohesion

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

Fig. 11 : Types of module cohesion

Functional Cohesion

- A and B are part of a single functional task. This is very good reason for them to be contained in the same procedure.

Sequential Cohesion

- Module A outputs some data which forms the input to B. This is the reason for them to be contained in the same procedure.

Procedural Cohesion

- Procedural Cohesion occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed.

Temporal Cohesion

- Module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed in the same time-span.

Logical Cohesion

- Logical cohesion occurs in modules that contain instructions that appear to be related because they fall into the same logical class of functions.

Coincidental Cohesion

- Coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another.

Relationship between Cohesion & Coupling

If the software is not properly modularized, a host of seemingly trivial enhancement or changes will result into death of the project. Therefore, a software engineer must design the modules with goal of high cohesion and low coupling.

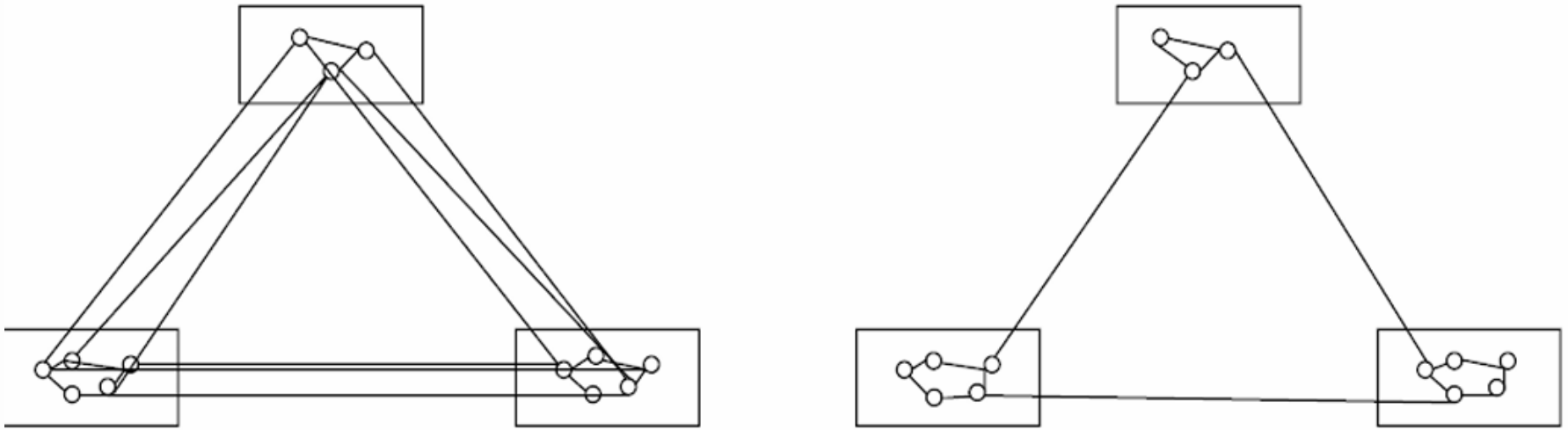


Fig. 12 : View of cohesion and coupling

Architectural Design

- The software needs an architectural design to represent the design of the software.

- IEEE defines architectural design as "the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system."

Data centered architectures:

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

Data centered architectures:

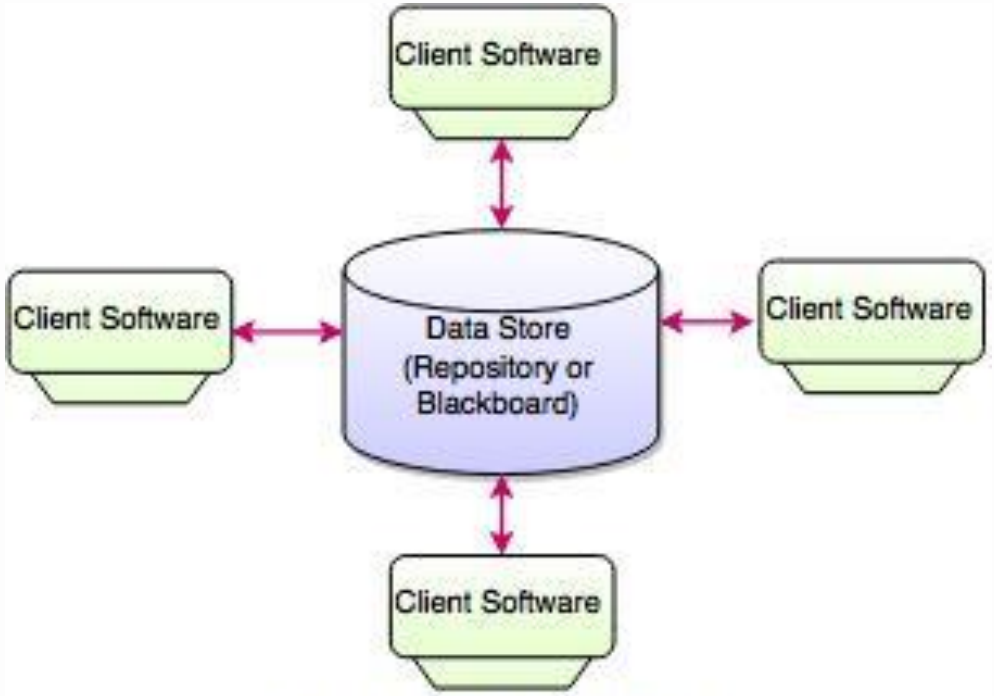


Fig. Data Centered Architecture

Data flow architectures

- This kind of architecture is used when input data is transformed into output data through a series of computational manipulative components.
- The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by lines.
- Pipes are used to transmitting data from one component to the next.
- Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighboring filters.
- If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.

Data flow architectures

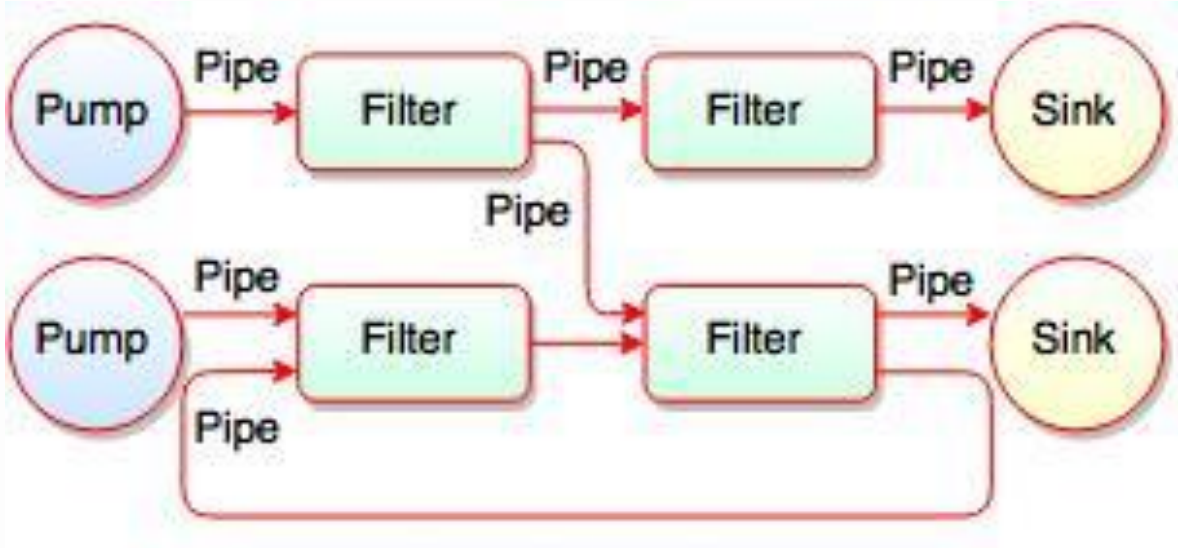
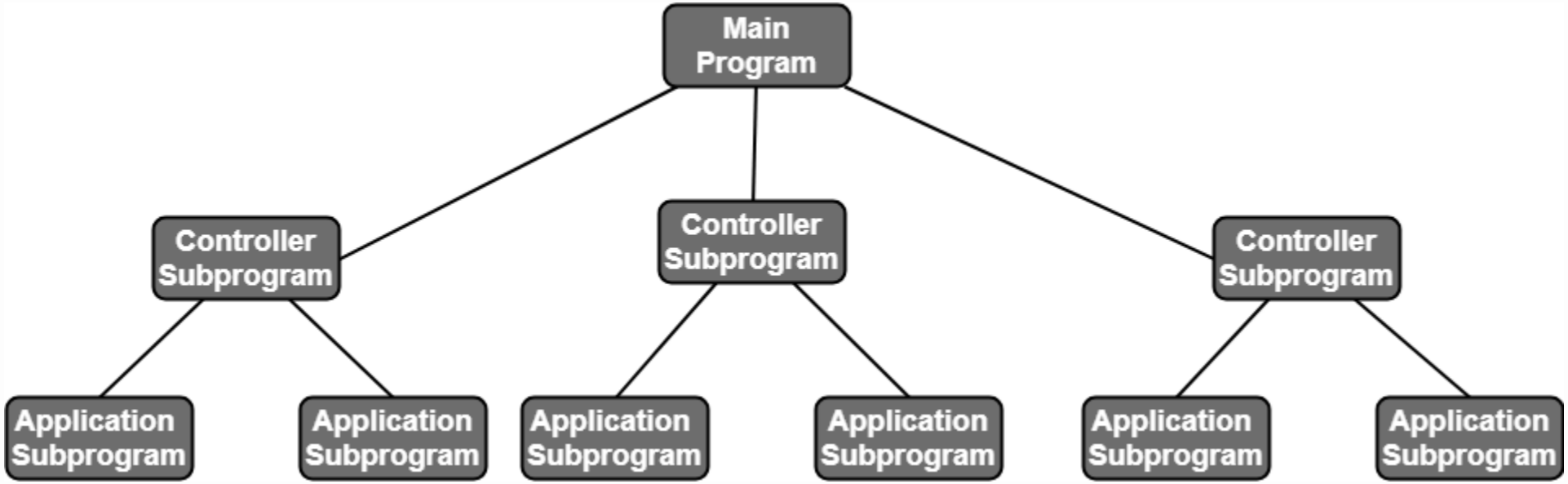


Fig. Pipes and Filters

Call and Return architectures

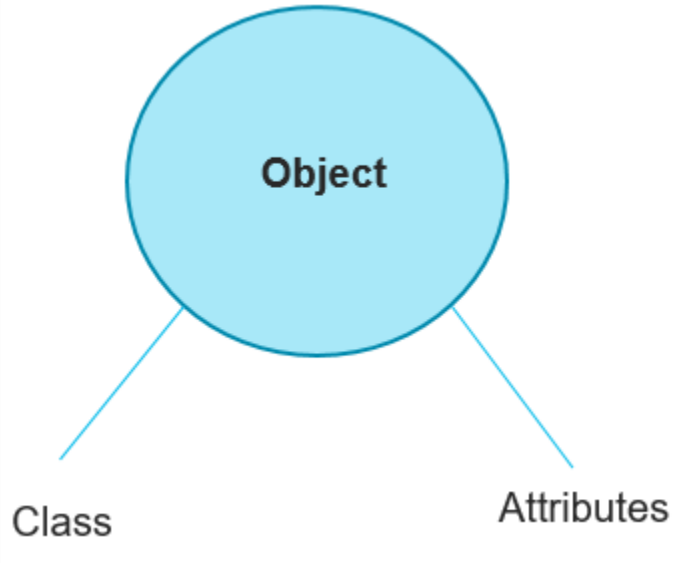
- It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category.
- **Remote procedure call architecture:** This components is used to present in a main program or sub program architecture distributed among multiple computers on a network.
- **Main program or Subprogram architectures:** The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.

Call and Return architectures



Object Oriented architecture

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



Layered architecture

- A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.
- At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing(communication and coordination with OS)
- Intermediate layers to utility services and application software functions.
- One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organisation for Standardisation) communication system.

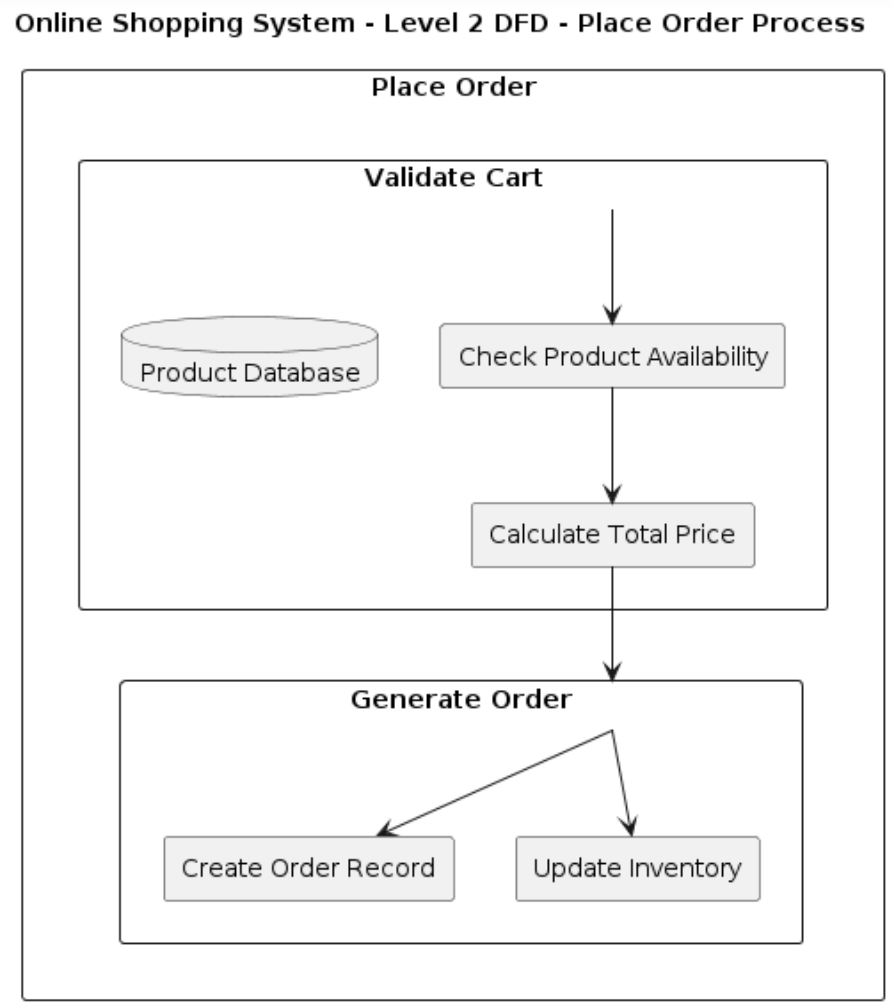
Layered architecture

Detailed Design Transaction Transformation

- **Transaction Transformation is a structured method that converts a Data Flow Diagram (DFD) characterized by a central "transaction" center into a functional, hierarchical structure chart.**

- **It identifies a transaction center a bubble that triggers one of several action paths and maps it into a high-level module that controls action-specific modules.**

Detailed Design Transaction Transformation



Aspects of Transaction Transformation:

Purpose: It maps a DFD with transaction flow (where data triggers specific, distinct paths of action) into a module architecture.

Transaction Center: Identifies a central processing bubble in the DFD that evaluates input data and chooses one of several action paths (e.g., in a banking system, selecting "Deposit," "Withdraw," or "Balance Inquiry" from a main menu).

Structure Chart Mapping:

Main Transaction Module: Created for the center, which dispatches data to the correct action module.

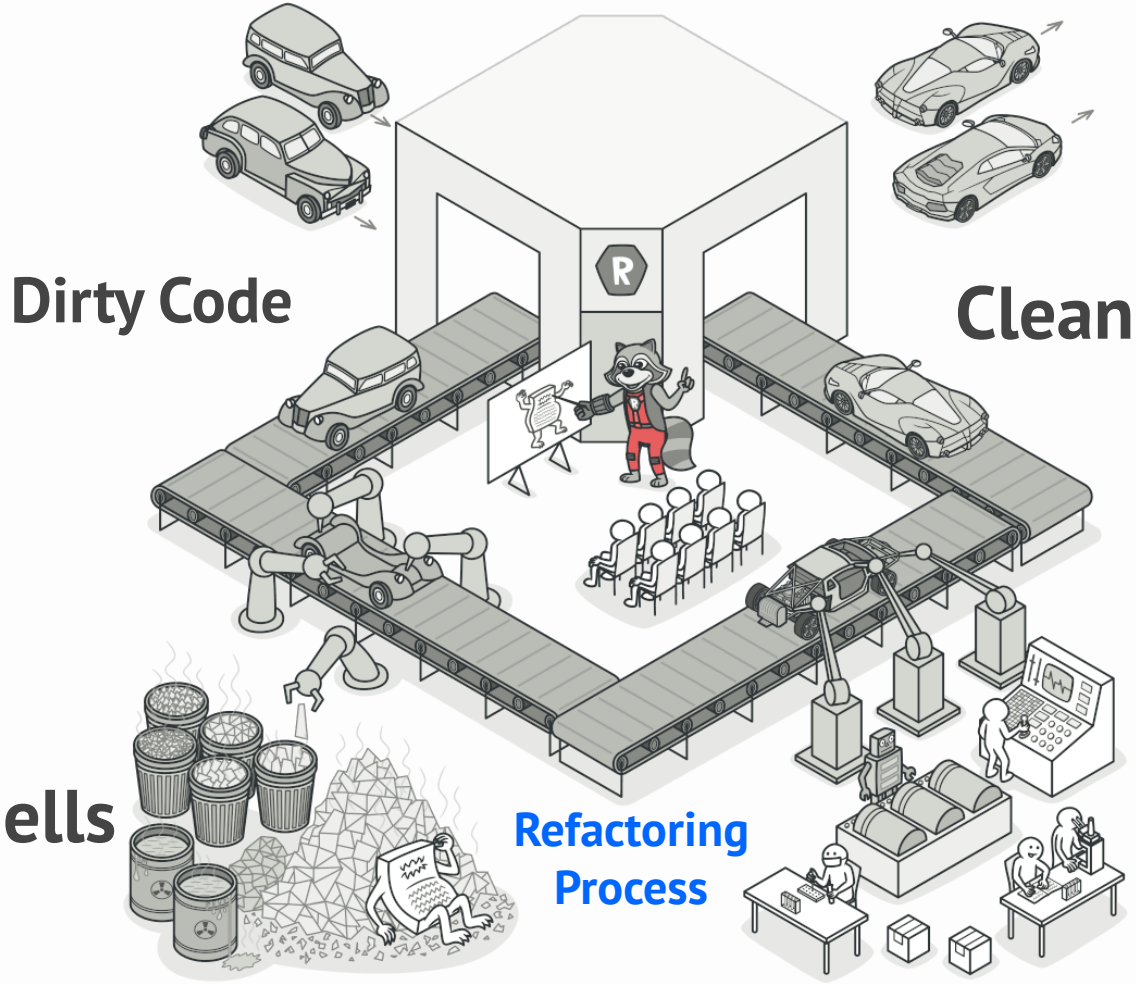
Action Modules: Derived from the paths following the transaction center.

Input Modules: Handle the incoming transaction data.

Refactoring of designs

- Refactoring is a systematic process of improving code without creating new functionality.
- Refactoring transforms a mess into clean code and simple design.
- Refactorings are code transformations that improve a system's maintainability without affecting its behavior.
- It refers to modifications in the code, such as splitting a function into two, renaming a variable, moving a function to another class, or extracting an interface from a class, among others.

Refactoring of designs



Refactoring of designs

- Dirty code is result of inexperience multiplied by tight deadlines, mis management, and nasty shortcuts taken during the development process.
- Code smells are indicators of problems that can be addressed during refactoring. Code smells are easy to spot and fix, but they may be just symptoms of a deeper problem with code.
- Clean code is code that is easy to read, understand and maintain. Clean code makes software development predictable and increases the quality of a resulting product.

Refactoring of designs

- Performing refactoring step-by-step and running tests after each change are key elements of refactoring that make it predictable and safe.

- Refactoring techniques describe actual refactoring steps. Most refactoring techniques have their pros and cons. Therefore, each refactoring should be properly motivated and applied with caution.

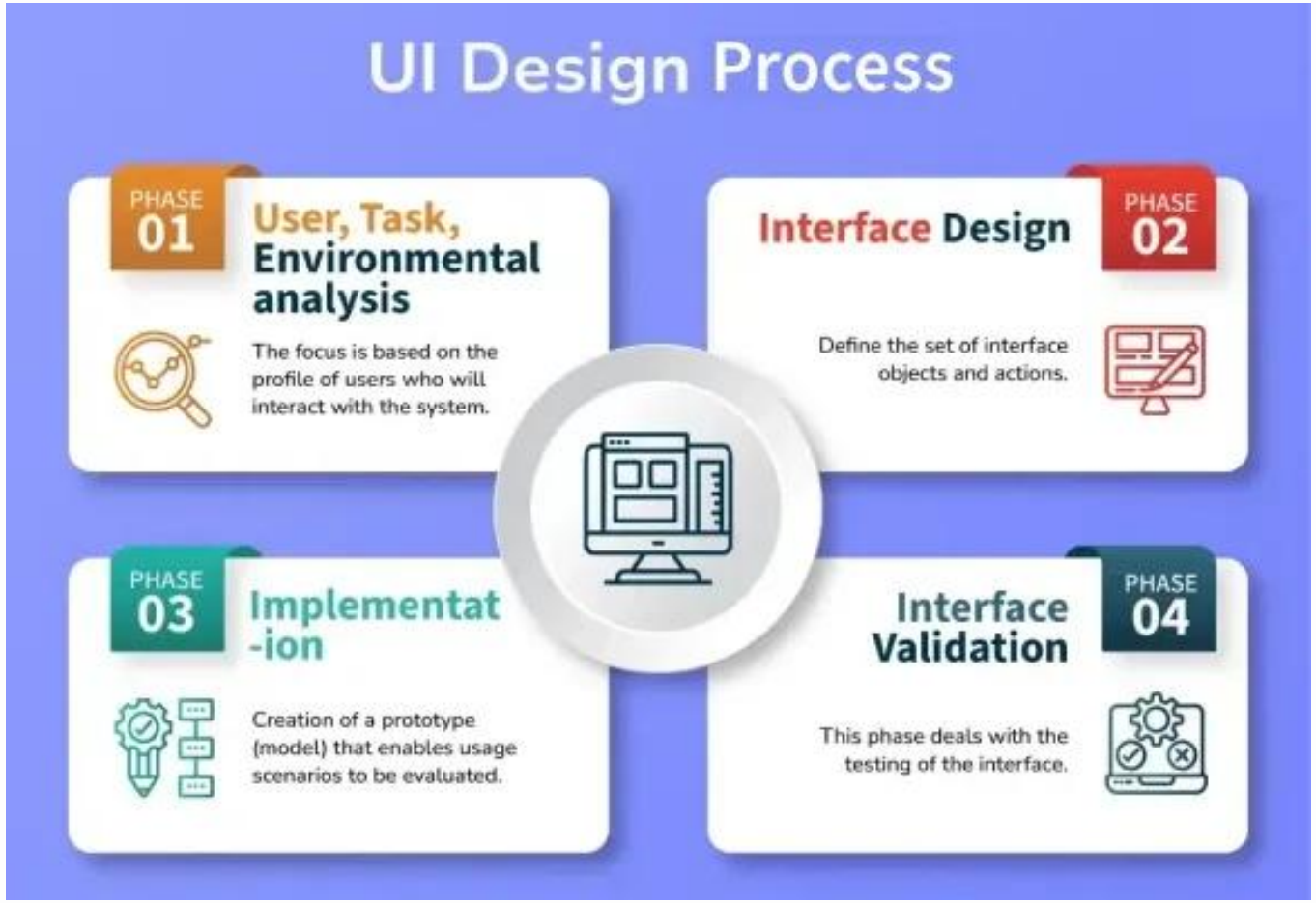
Object oriented Design

- Object-oriented design (OOD) is a programming technique that solves software problems by building a system of interrelated objects.
- It makes use of the concepts of classes and objects, encapsulation, inheritance, and polymorphism to model real-world entities and their interactions.
- A system architecture that is modular, adaptable, and simple to understand and maintain is produced using OOD.
- Encapsulation, Abstraction, Inheritance, Polymorphism, …

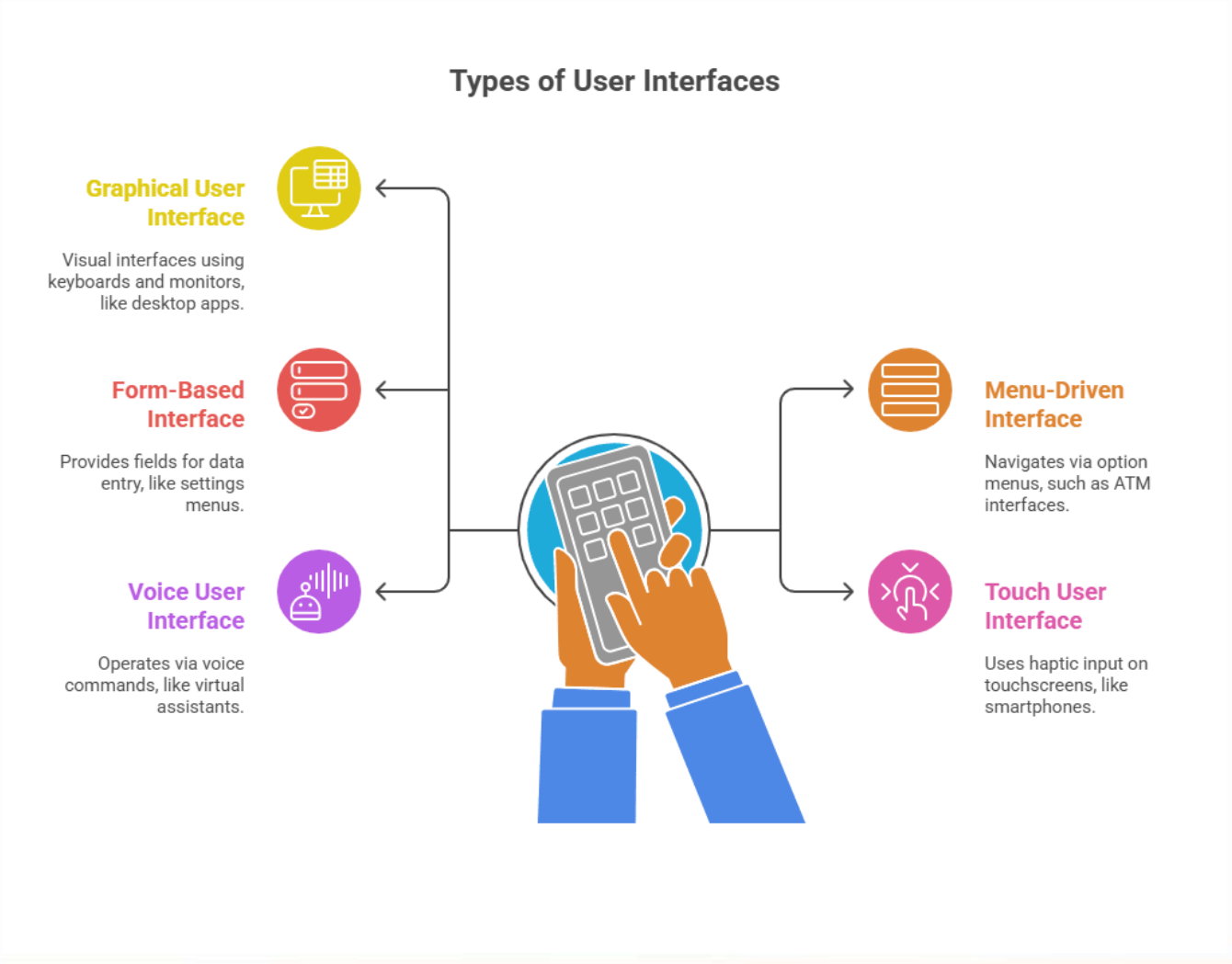
User-Interface Design

- User Interface (UI) Design shapes how users interact with digital systems like websites and apps.
- It creates easy-to-use, attractive interfaces that improve usability and show a brand's style. This article covers UI design basics, types, importance, and how it engages users.
- **Easy to Use:** UI design makes apps and websites simple to navigate with clear buttons, menus, and layouts.
- **Attractive Look:** Creates visually appealing designs with colours, fonts, and images that match a brand's style.
- **User Actions:** Handles inputs like clicks or taps and shows outputs like results or visuals.

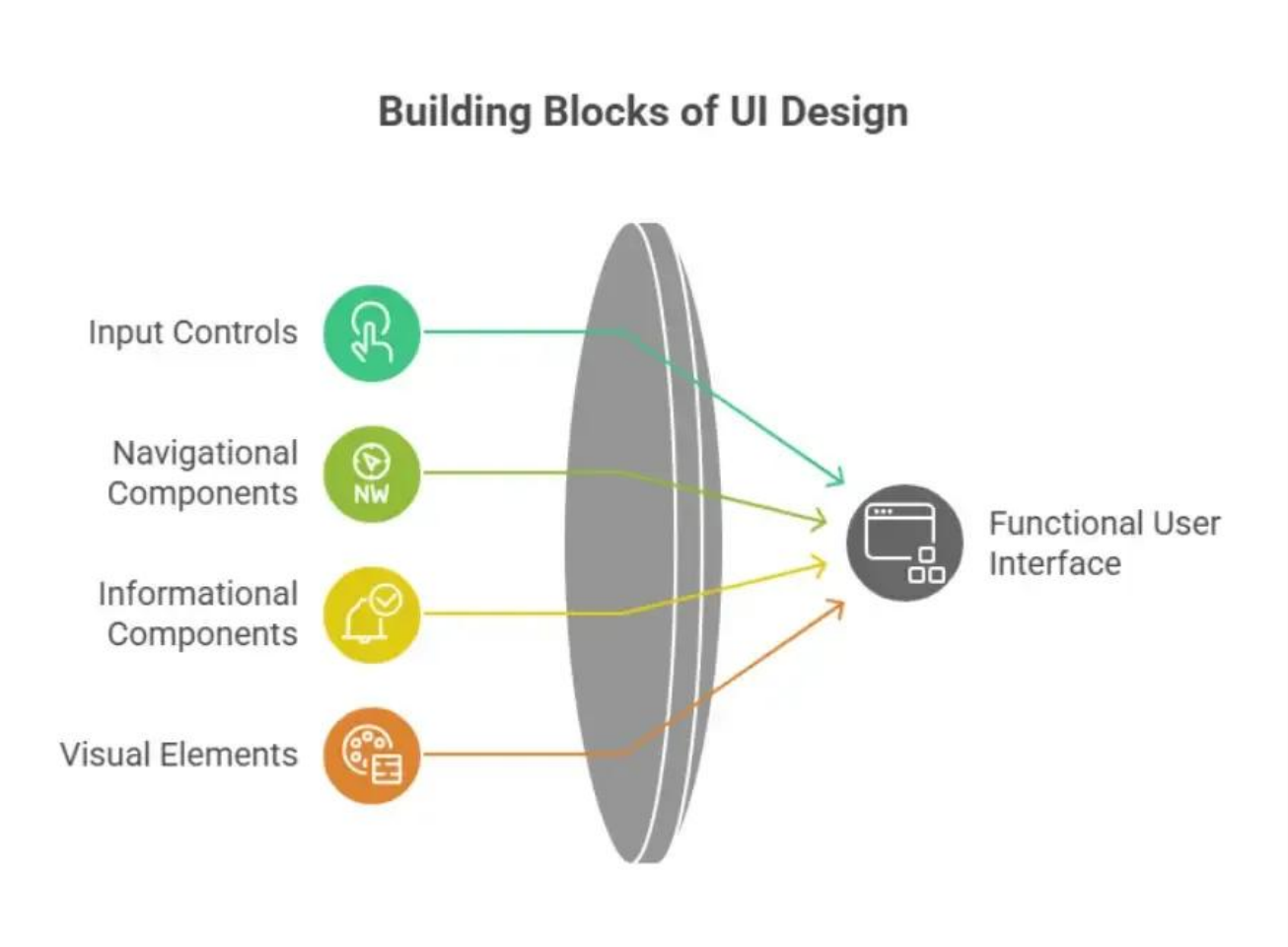
User-Interface Design



Types of UI Design



Components of UI Design



Components of UI Design

User Interface(UI)	User Experience(UX)
UI focuses on the quality of the user's contact with the product	UX centres around the intent and functionality of the product.
UI comprises more creative design elements linked to the look and feel of the user's experience	UX involves components like market research and understanding consumer needs.
UI is more particularly focused on the design of the end product	UX are concerned with managing the overall project from ideation through development and delivery


thank
you