# Unit 3 – Software Design

## Introduction

## Unit 3 Outcomes

- Discuss software design and its activities.
- Understand what makes a good software design.
- Understand software design strategies such as:
- Top-down and bottom-up Approaches

# Unit 3 Outcomes

- Understand the meaning of functional independence, coupling, and cohesion.
- Appreciate the application of DFDs and structure charts in software design.
- Apply Unified Modelling Language (UML) for Object-oriented design (OOD).
- Understand basics of software development.

# Software Design

- Software design transforms the requirements into a form that can be implemented using a suitable programming language.
- Software design concentrates on the solution domain.
- Design is very important when the system gets bigger with many developers.

# Examples

- **If you are ordering pizza online, for cash on delivery, the charge is Rs. 300. For a credit card, the charge is Rs. 250. If your account has coupons of a minimum of Rs. 50, then the discount can be in the range of Rs. 50 to Rs. 100. If the order is delayed by 5 minutes the pizza is given free.**

**This problem needs an elaborated software design**

- **Some more examples:**
- **Design drawings of buildings**
- **Models for building or simulations of a machine**
- **Design plan of a trip: Date, place, hotel, time, important places to visit, etc.**

# Software Design Activities

- **Preliminary or High-Level Design**
- **Detailed Design**

| |
|---|
| **High-Level Design: College Management Software**<br>Admission, Exam<br>Departments, Laboratories<br>Library<br>Hostel<br>Sports Complex, Cafeteria |
| **Detailed Design:** Functions and data such as:<br>Admission: Name, Roll No, Marks, Eligibility, Age, Address…<br>Library: Roll No, ID, Name, Books_Issued |

# Unit 3 – Software Design

## Software Design - Introduction

# Software Design

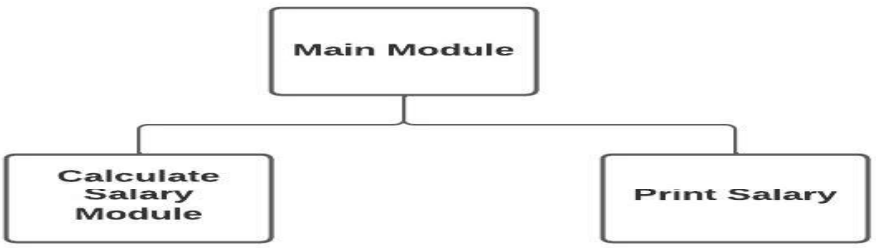

- **Preliminary Software Design**
- **Detailed Design**

# High Level Software Design

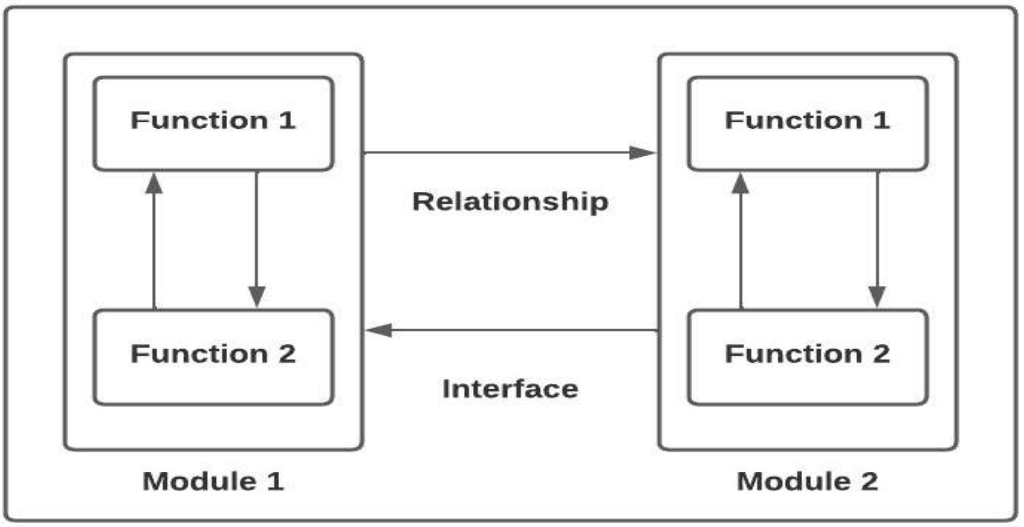
- Modules
- Control relationships
- Interfaces

# Module

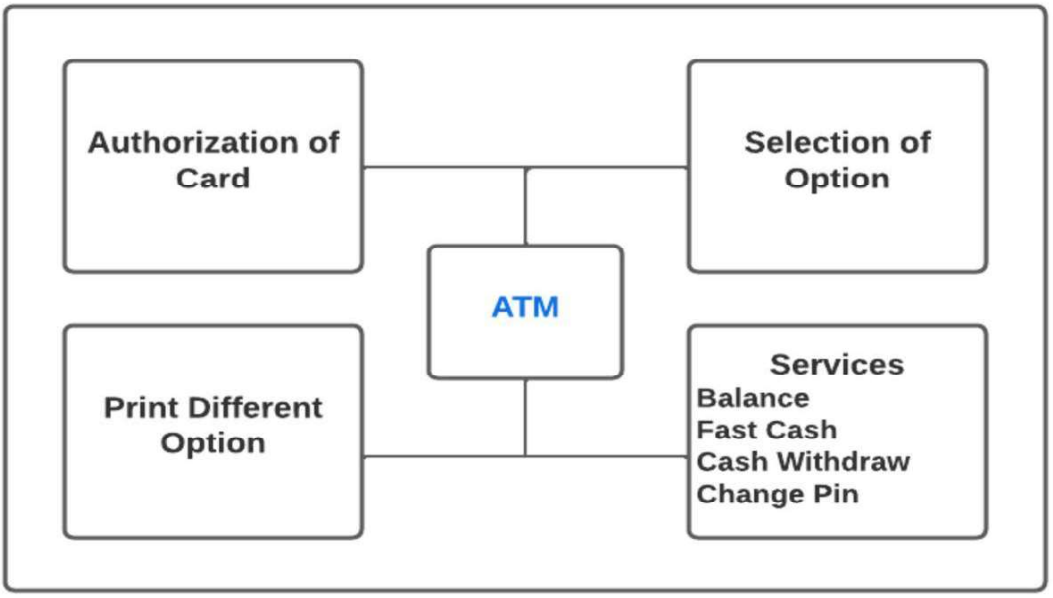
- Module is a combination of many functions and data structures.
- Modules divide the problem into subproblems using divide and conquer.
- Modules are almost independent



# High Level Design




# Example



# Unit 3 – Software Design

## Detailed Software Design

## Detailed Software Design

- Program Units
- Allocation of Functions to Units
- Data flow between Units
- Control Between Units
- Data Structures
- Algorithms

## Detailed Software Design

- Data Packages
- Data Visibility or Scope
- Operating System

## Detailed Software Design

- Hardware-related Design
- Software-related Design
- Security- related Design
- Performance-related Design
- Internal-Communication related Design

# Data Dictionary

- Data dictionary is a tool to describe the detailed design.
- Data Dictionary stored data about data, known as metadata
- In relational database data dictionary may include the following

❑ Names of tables, Field width, Conditions, Primary keys, foreign keys etc.

# Data Dictionary

| Name of Attribute | Required (Yes/No) | Format | Maximum Field Size | Location Details |
|---|---|---|---|---|
| Student Name | Yes | Text | 40 | C://data//name.xls |
| Roll No | Yes | Number | 3 | C://data//no.xls |
| Date of Birth | Yes | Date | 10 | C://data//dob.xls |
| Credit Card Number | No | Text | 20 | C://data//creditno.xls |

# Good Software Design

Good Software Design

- Correctness
- Understandability
- Efficiency
- Maintainability
- Completeness
- Consistency

# Good Software Design

- **Correctness:** Performs tasks as per specification
- **Understandability:** Uses meaningful names, well-divided modules with a neat hierarchy. E.g., tree or layering modularity
- **Efficiency:** It is cost-effective for using resources

# Good Software Design

- **Maintainability:** Flexible for change requests
- **Completeness:** All the data objects, interfaces, modules, relationships must be covered
- **Consistency:** All the elements in the software behave and look uniformly.

# Software Design Principles

Software Design Principles

- Divide and Conquer
- Abstraction

# Software Design Principles

- Modularity
- Reusability

# Software Design Principles

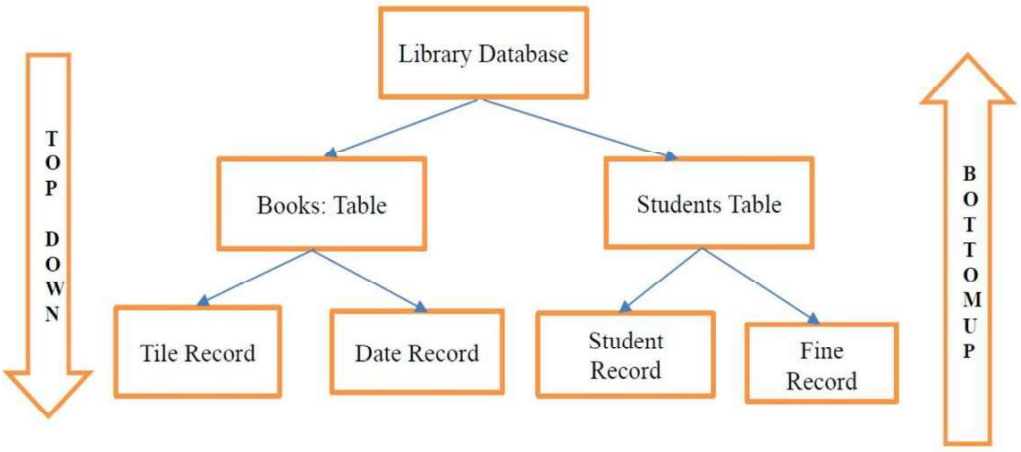
- Flexibility
- Portability
- Testability

# Software Design Strategies

Software Design Strategies

| Level-Oriented Software Design | |
|---|---|
| Top-Down Approach | Bottom-Up Approach |
| Function-oriented Software Design | Object-oriented Software Design |

# Level-Oriented Software Design

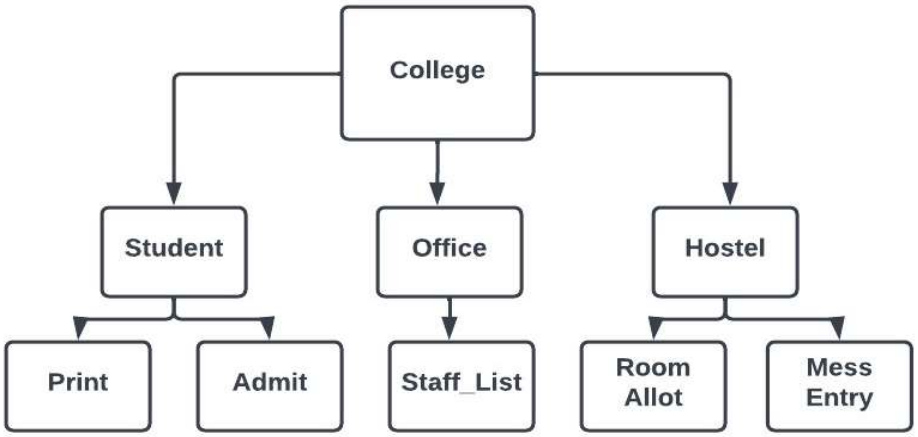



# Function-oriented Software Design

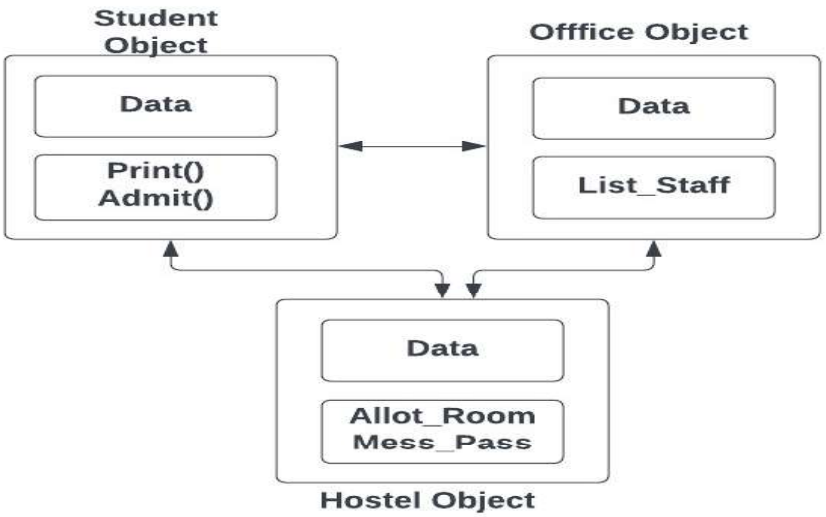
- Top-down design approach
- Three Steps:

1) Design of data flow: DFDs
2) Structure division: Structure Charts
3) Detailed design: Data Dictionary

# Top-down design




# Bottom-Up Design



# Object-oriented Software Design

- Bottom-up design approach
- Importance is given to the data
- Uses object attributes and functions
- Uses class, objects, and relationships etc

# Top-down Vs Bottom- up Software Design

| Top-down approach | Bottom-up Approach |
|---|---|
| Division of problem is based on functions or procedures- Function oriented | Division of problem is based on classes: Object-oriented |
| Data is stored in centralized memory and shared in functions | Data is distributed to objects |
| Begins by identifying scenarios and functions. | Begins by classes and objects |
| Languages used: C | Languages used: Java, C++ |

However, in practice, both the top-down and bottom-up design strategies are combined and used together

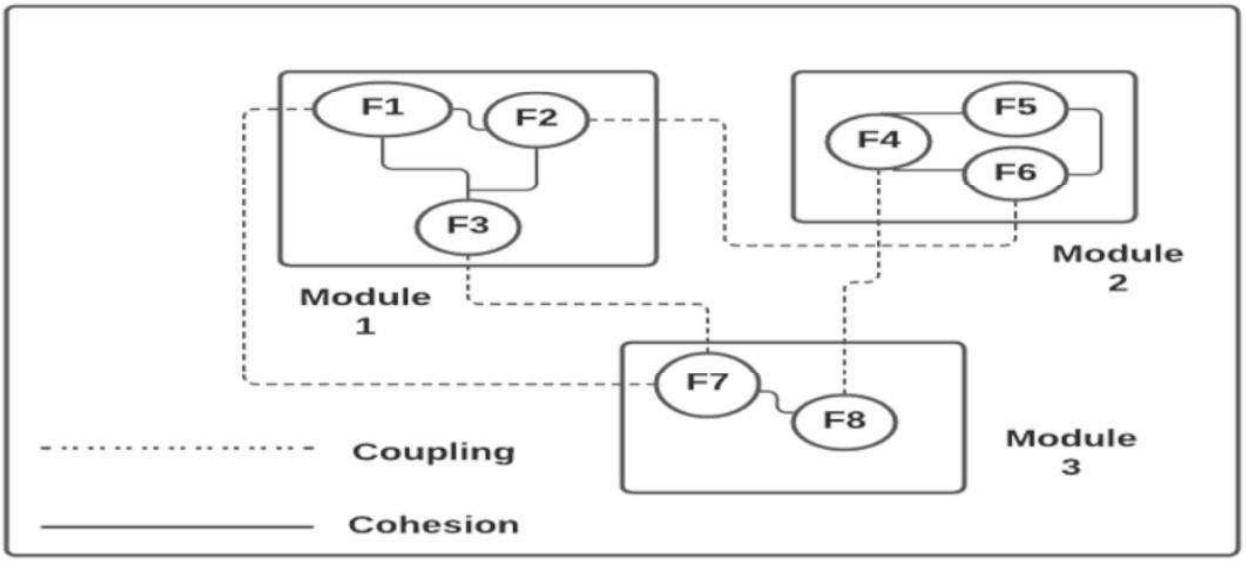
# Software Design Principles

## Cohesion and Coupling

## Software Design Principles

- Cohesion: Defines how strongly the elements are related to each other in a module.
- Coupling: Defines dependency of modules on each other.

# Cohesion and Coupling




# Examples

- High Cohesion

| Student |
| --- |
| Name<br>RollNo |
| PrintRecord(Name, RollNo)<br>printResults(RollnNo)<br>printAttendace(RollNo) |

- Low Cohesion

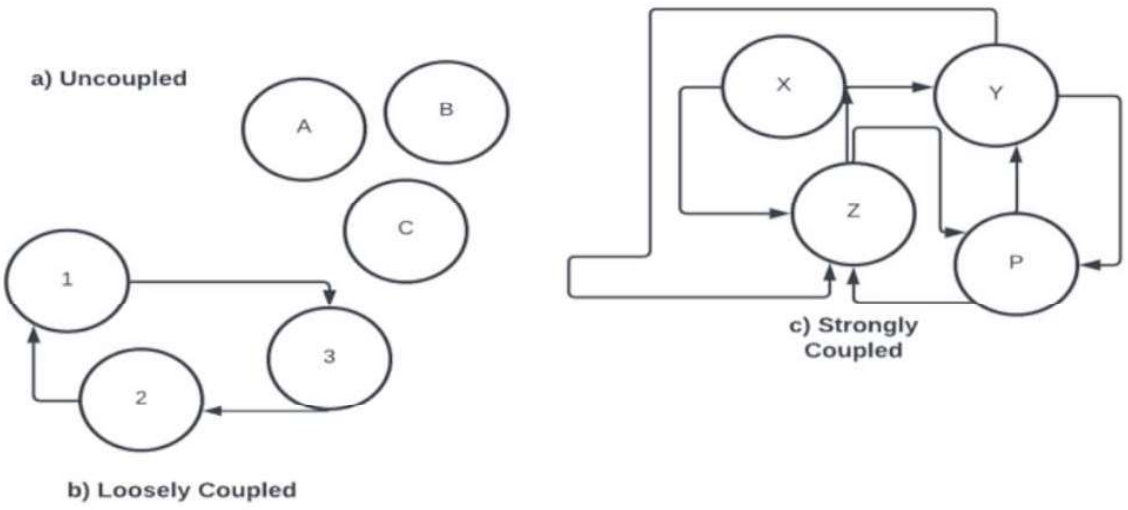
| Student |
| --- |
| CheckEmail() |
| Print()<br>PrintStafflist()<br>CalculateFees()<br>CheckName() |

Higher Cohesion is Better!

# Coupling

- Coupling is interdependence of modules or components
- If two modules exchange large amount od data then they are highly coupled
- Lower/Loose Coupling is Better

# Coupling



# Example - Coupling

- Coupling

| Login Module |
| --- |
| Back End Module |
| Exception Module (For Incorrect Password etc.) |

- Lower Coupling is Better!
- Higher or tight Coupling can be based on:
- Data, control, code, content, etc.



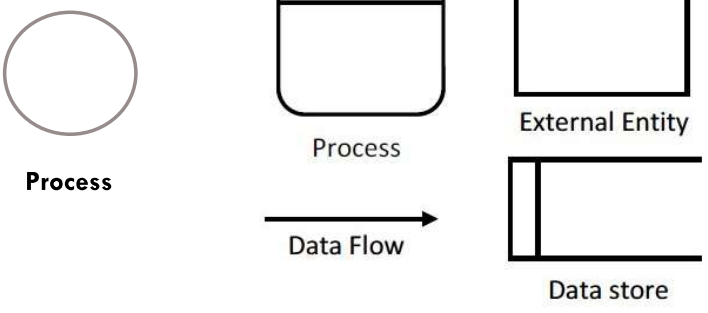
# Data Flow Diagram (DFD)

# DFD- Learning Outcomes

- Understand what is a DFD
- Draw the DFDs for a given problem
- Understand 0-, 1- and 2-level DFDs

# DFD

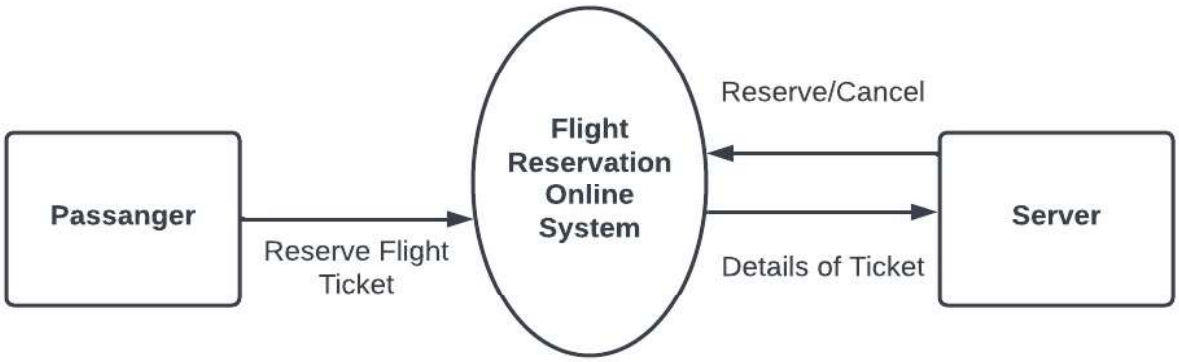
- DFD shows the system requirements in a graphical manner.
- DFD expresses the functions or activities in the system and data flow. Rounded rectangles, rectangles, circle and arrows are the symbols used in DFDs.



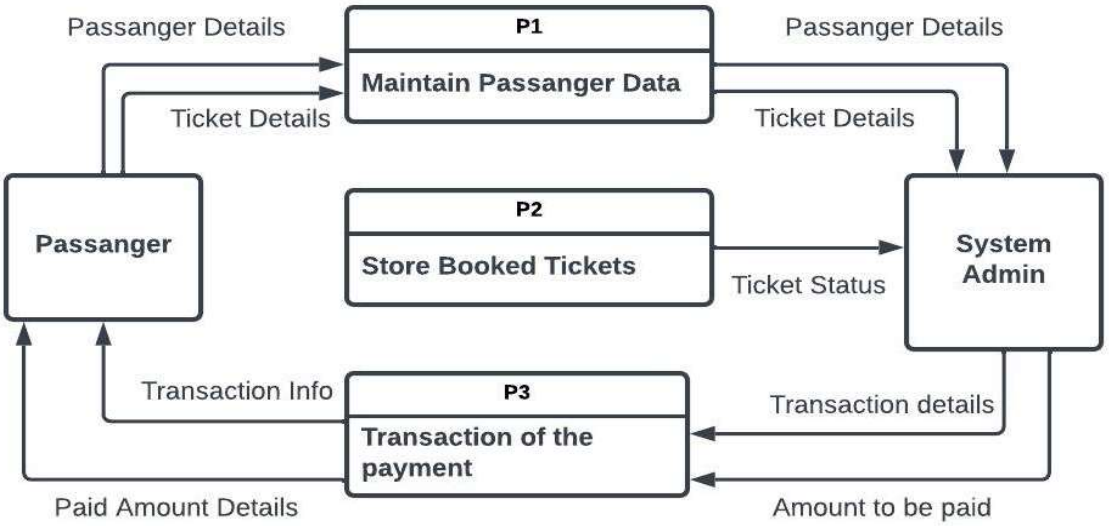
# DFD Hierarchy

- **0-Level DFD/Context Diagram:** Presents an overview of the system and its interaction with the rest of the world
- **1-Level DFD:** Presents a more detailed view of the system than context diagrams, by showing the main sub-processes and stores of data
- **2-level DFD:** Certain elements of any dataflow diagram are decomposed into a more detailed model

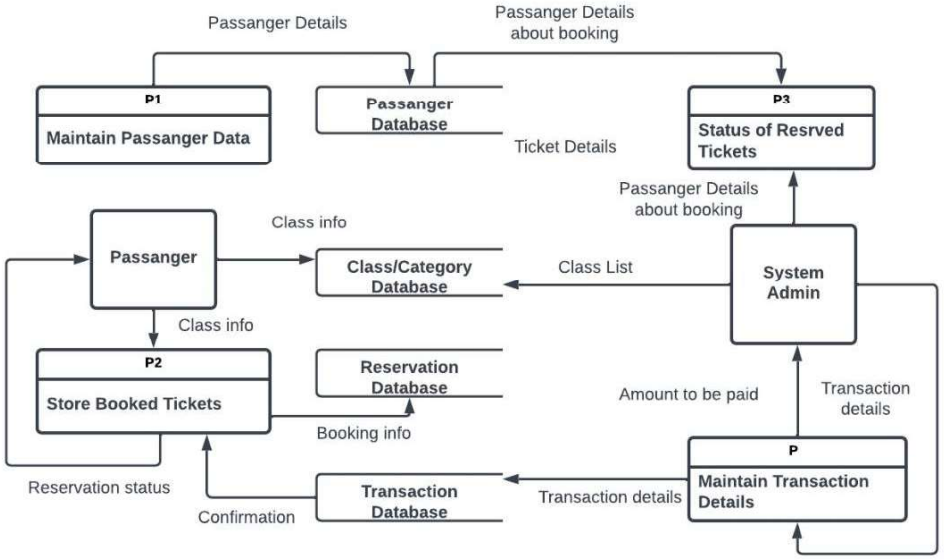
# 0-Level DFD/Context Diagram



# 1-Level DFD




# 2-Level DFD



# Structure Charts

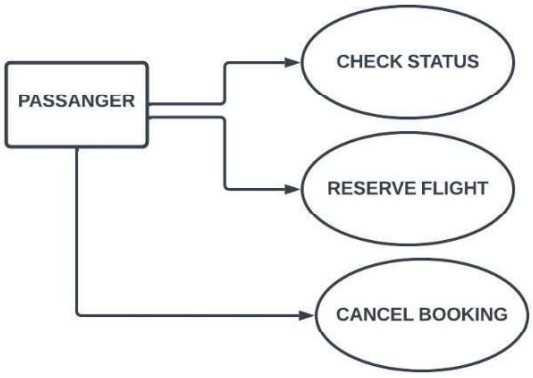
## Structure Charts

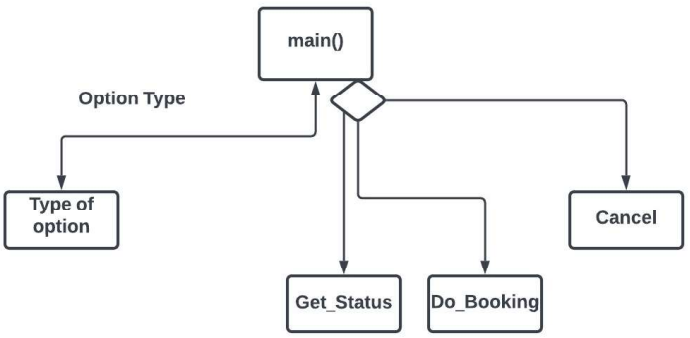
- Used in function-oriented design.
- Structure charts are derived from the DFDs.
- Decomposes the problem into subproblems to show the hierarchical relationship.
- Example: Organization Chart

# Symbols in Structure Charts

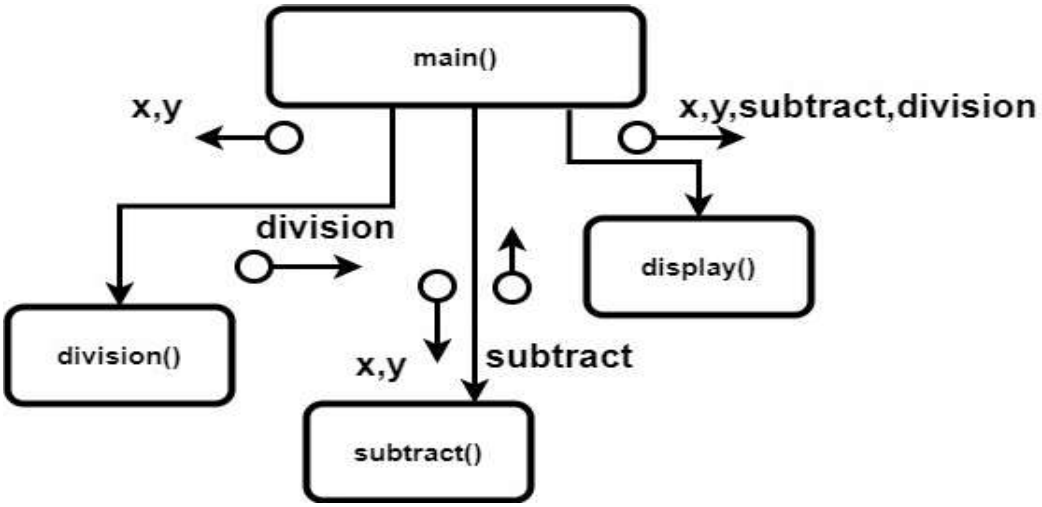
- Modules are represented by rectangular boxes
- Arrows with annotations are used to pass control from one module to another with a given direction
- Library modules are represented by a rectangle with double edges.
- Diamond for the selection command and loops are represented using the repetition around the control flow.

# DFD and Structure Charts

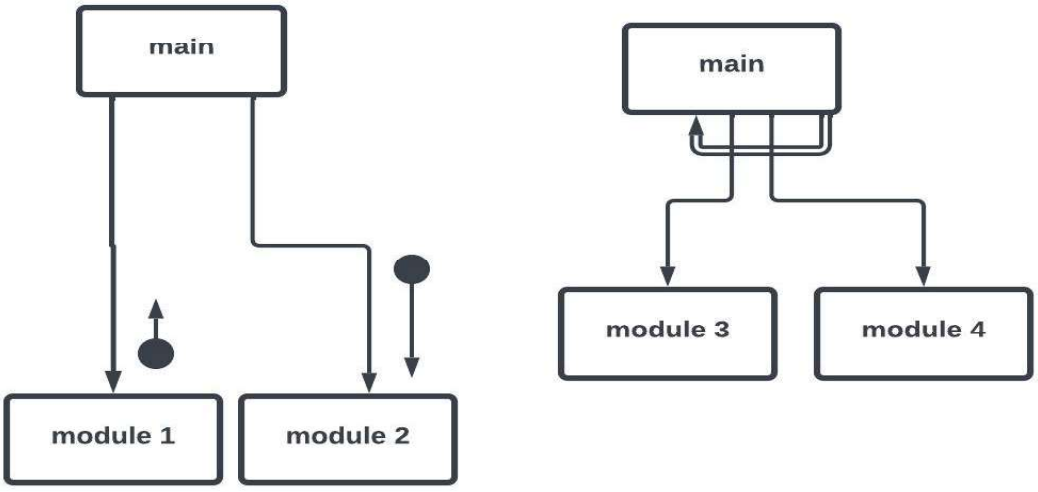


# Structure Charts




# Structure Charts



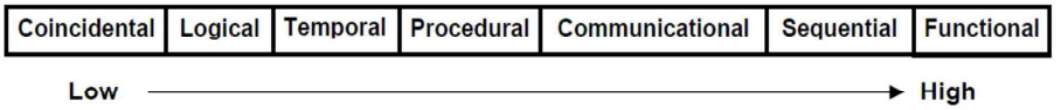
# Cohesion

## Learning Outcomes

- State the meaning of cohesion
- Classify different types of cohesion

# Cohesion

- Cohesion is a primary characteristic of a module
- Cohesion is a measure of functional strength of a module
- Higher Cohesion is Better
- Classification of Cohesion:




# Coincidental and Logical Cohesion

- Coincidental cohesion means parts of a module are combined randomly or arbitrarily
- Example: Print_Adhar_details(), File_Error(), Calculate_Balance() functions in a single module
- Logical cohesion: A module is said to be logically cohesive if all elements of the module perform similar operations
- Example: print() function in a module printing student attendance, teacher attendance, student marks, student details

## Temporal and Procedural Cohesion

- Temporal cohesion all the functions of a module are executed in the same time span.
- Example: Set of functions for start, shutdown, sleep, initialize of any process in a computer
- Procedural cohesion: A module is said to have procedural cohesion if all the functions in it are part of the same algorithm working towards a common goal.
- Example: Enter user id, password, captcha are parts of a common authorization goal

## Communicational & Sequential Cohesion

- In communicational cohesion all the functions refer to or update the same data structure
- Example: add(), delete(), print(), modify() functions working on a database STUDENT
- In sequential, cohesion, the elements of a module form a part of a sequence. Here the output of one element is input to next
- Example: insert a card→ enter a pin→ authorize functions used in ATM

# Functional Cohesion

- Function cohesion means different elements of a module work together to achieve a single task.
- Example:

  Select cabin/business class/ choose childcare seat/window seat preferences etc. are the functions related to the seat_allotment module in the Airline_Ticket_reservation system
- Functional Cohesion is the most desirable

# Summary

- Cohesion reduces complexity in a module.
- Cohesion improves maintenance as only a few modules get affected.
- Cohesive set of operations in a module increases reusability.
- A module with a single element and function is perfect cohesion. But this may be too narrow or complicated.
- Thus, a perfect balance between cohesion and coupling is necessary.

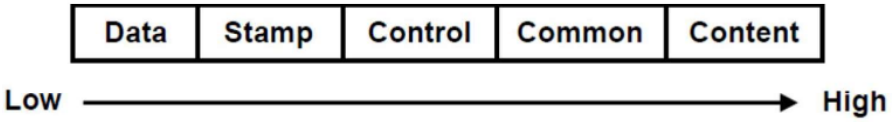
# Coupling

## Learning Outcomes

- State the meaning of coupling
- Classify different types of coupling

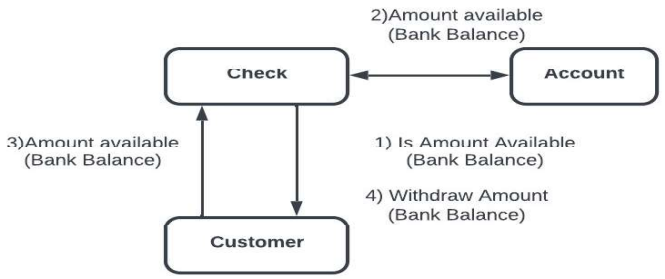
# Coupling

- Coupling between two modules is a measure of the interaction between the two modules
- Coupling depends on the complexity between the modules
- Lower Coupling is Better
- Classification of Coupling:




# Data and Stamp Coupling

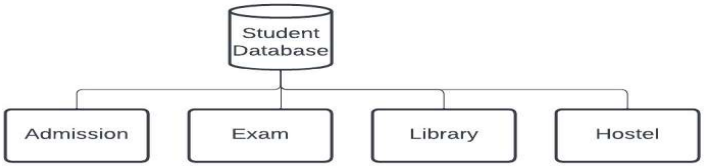
- In data coupling two modules communicate using a common parameter.




- In stamp coupling, two or more modules use composite data. Example: Structure data type in C

# Common and Content Coupling

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




- In content coupling, more than one module share code, Example: Branch from one module to another

```
void menu(){
 { add(); subtract(), mult(), div(); print()}
```

# Control Coupling

- Control coupling exists between two modules if data from one module is used to direct the order of instructions executed in another.
- Example:

```
void draw(char command[])
{
  if(command==“circle”)
Draw_circle()
else
Draw_square()
}
```

# Conclusion

- Coupling reduces complexity in a module
- Coupling defines the boundary of modules and identifies the connection
- Lower the coupling, better is the performance of a system

# Learning Outcomes

- Discuss when the module is called functionally independent.
- Understand the importance of functional independence in software design.
- Discuss modularity in software design.

# Functional Independence

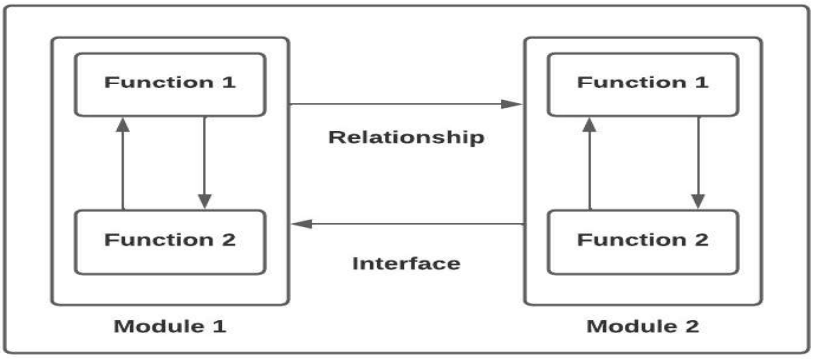
- A module with high cohesion and low coupling is said to be functionally independent of other modules
- A functionally independent module performs a single task without much interaction with other modules.
- Functional independence is the key to good software design

# Functional Independence

- **Error Isolation: With less interaction, any error existing in a module would not directly affect the other modules**
- **Scope of Reuse: A cohesive module can be easily taken out and reused in different programs**
- **Understandability: Reduces complexity of the design and modules can be understood in isolation**

# Modularity

- **Modularity is a decomposition of programs into smaller programs with a standard interface.**
- **Example: Modules→Packages→Classes→Data→Functions→Other classes**



# Modularity

- Easy to understand the system.
- System maintenance is easy, as any changes can be accommodated.
- A module can be used many times as their requirements.

  No need to write it again and again!