Unit 3 – Chapter 7

Software Design and Development

Pre-requisite(s)

Knowledge of software requirements, software development life cycle

Unit Outcomes

At the end of this chapter, the student will be able to:

- Discuss software design and its activities
- Understand what makes a good software design
- Understand software design strategies
- Understand the meaning of functional independence, coupling, and cohesion
- Appreciate the application of DFDs and structure charts in software design

7.1 Introduction to Software Design

The design phase begins after the completion of the requirement analysis. While requirements specify *what* the software is supposed to do, design specifies *how* to develop the system so that it meets the requirements.

Software design deals with transforming the customer's requirements into a form that can be implemented using a programming language. Software design concentrates on the solution domain.

Designing is an important activity as:

1. It provides the basic framework that guides how the program code should be written and how personnel are assigned to tasks.
2. Design errors are often costlier than coding errors. They take more time to detect and correct. Therefore, the design provides a basis for monitoring progress.
3. A poorly designed software product is often unreliable, inefficient, and not maintainable.

4. Design becomes important when the systems become larger, and many developers are involved.

7.2 Software Design and Activities

Design activities can be broadly classified into two activities:

- Preliminary or high-level design: This includes the identification of different modules. When functions in two or more modules are called then there is a control relationship amongst the modules. Also, when the data is exchanged amongst different modules, then there is an interface. The high-level design identifies such relationships and interfaces. The outcome of the high-level design is called the program structure or software architecture.
- Detailed Design: During detailed design, the different modules' data structure and algorithms are designed. The outcome of the detailed design stage is usually known as the module-specification document.

The following items are designed during the design phase:

1. Modules
2. Control relationships among the identified modules.
3. Interfaces among the modules
4. Data structures of individual module
5. Algorithms to implement each module

7.3 Characteristics of a Good Software Design

Characteristics of a good software design are as follows:

1. Correctness: A good software design should correctly implement all the functionalities identified in the SRS document.
2. Understand-ability: Software design should be easy to understand
3. Efficiency: Resources such as time, space, and cost should be optimized in a design.
4. Maintainability: The design should be simple and easy to maintain.

5. Completeness: The design should have all components like data structures, modules, external interfaces, etc.

6. Consistency: There should not be any inconsistency in the design.

7.4 Software Design Principles

Software design principles are concerned with providing means to effectively handle the design process's complexity. Following are the principles of software design:

1. Problem Partitioning or Divide and Conquer: It means breaking a problem into smaller bite-sized subproblems to reduce the complexity of the problem. For software design, the goal is to divide the problem into manageable pieces. However, these pieces cannot be completely independent of each other and must cooperate and communicate with each other to solve the problem. This added communication increases the complexity.

2. Increase Abstraction: An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation.

3. Modularity: Modularity specifies the division of software into separate modules which are differently named and addressed and are integrated later to obtain the completely functional software. A problem should be decomposed cleanly into modules that are mostly independent of each other. This reduces complexity greatly.

4. Increase Cohesion: Cohesion means grouping things that make sense together. Cohesion organizes your code, and it will make it much easier to find things, thus simplifying the system.

5. Reduce Coupling: In simple terms, coupling occurs when packages, modules, classes, or files are very interdependent. If one package has some changes to it or breaks, the whole system could be compromised due to interdependence. Hence good software should have as independent components as possible. This will have an easier time debugging. That is a better design.

6. Increase Reusability: A good design would always ensure re-usability as this is very efficient.

7. Design for Flexibility: A good design should be flexible and anticipate changes to the system.
8. Design for Portability: While designing a system keep in mind that it may be used on a different platform or device in the future. Hence portability becomes important.
9. Design for Testability: Designing for testability becomes very important in large systems with large codebases.

7.5 Software Design Methodology and Strategies

Software design methodology presents the design process systematically. The methodology consists of design principles, rules, and a sequence of activities in the design process. The design methodology expresses the conceptual model of a design using diagrams. The design methodology also includes available hardware resources, software support, existing working system, and existing data structures. The software methodology also mentions the order of activities with suitable diagrammatic notations. The systematic design developed is called structured design.

In structured design, the given problem is divided into sub-problems and each problem is handled individually using separate modules. These modules are then organized in a specific order to deliver an accurate solution. **The high cohesion and less coupling describe the better-structured software design.**

There are different decisions to be taken in making a structured software design. These decisions decide whether the system is hierarchical, or it is functionally modular or based on the levelling approach of the application. This is called software design strategy. Some of the commonly used design strategies are as discussed below.

7.5.1 Level-oriented Software Design:

Top-down and Bottom-up Approach

There are two approaches to level-oriented strategy: top-down and bottom-up design. In the first approach, a general design solution is divided into sub-solutions or subsystems. Each sub-system is further divided into sets of components or a collection of sub-systems. The division is continued till the last level of the system. Thus, the top-down design strategy starts with a solution as a whole design and then designs of several sub-systems and components are added in further steps. If the design is to be formed from scratch or ground level, then the top-down approach is preferred.

In the bottom-up design strategy, the lowest level subsystems and components are designed first. The bottom-up design process further combines them into the next abstraction level. This process is continued till a single system is formed that includes all the sub-systems and components. This approach is useful when already there is an existing software system, and a new system is to be formed based on it.

However, in practice, both the top-down and bottom-up design strategies are combined and used together. An example of top-down approach and bottom-up approach is depicted in Figure 7.1.

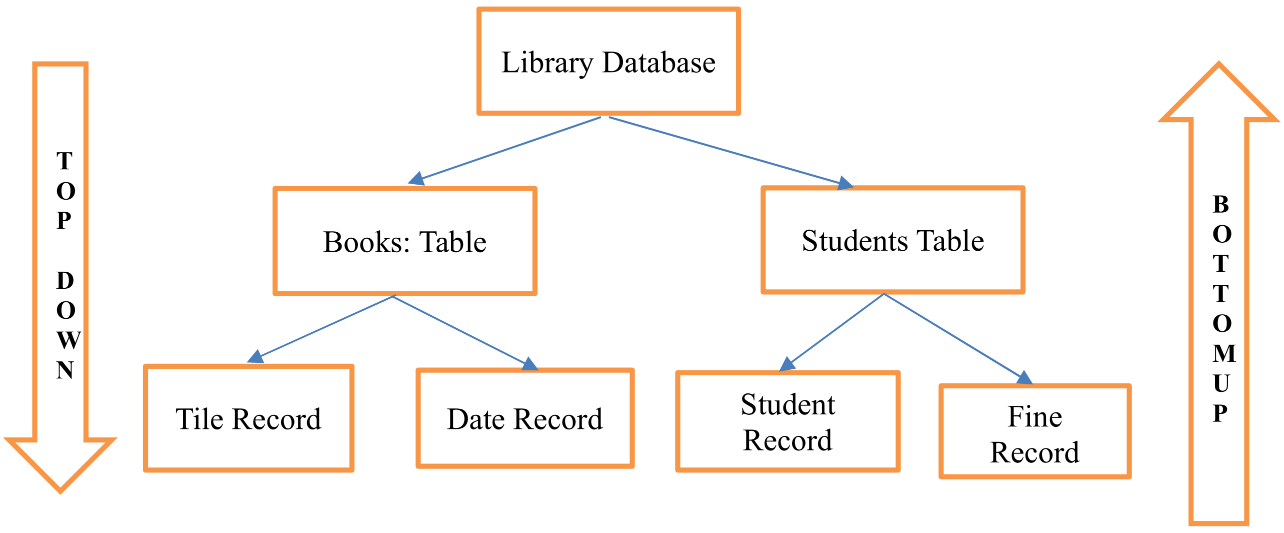


Figure 7.1 Top-down and Bottom-up Software Design for Simple Library Management

7.5.2 Function-oriented and Object-oriented Software Design Approach

An example of a top-down design approach is function-oriented software design. In the function-oriented design approach, the design is divided into a collection of units based on the function. Each unit is defined with a task and several units interact with each other. The functional design process consists of the following three steps.

a) Design of data flow: In this process, the processing of data is shown using DFDs

b) Structure division: In this process how the given project (system) is divided into different parts (sub-systems) is shown using structure charts.

c) Detailed design: In this process, the components, interface, relationships among the subsystems are described. Data dictionary tool is used here.

Object-oriented Software Design Approach

Object-oriented software design is a bottom-up design approach. Several modern programming languages such as Java, Python, C++, etc. use object-oriented programming where the importance is given to the data. The object-oriented software design approach is used to support this object-oriented programming. In function-oriented design, the importance is given to functions, whereas in object-oriented design, the data is stored in the object attributes and manipulated using functions. In object-oriented design, class is an abstract data type that calls the functions working on data. The important terms such as class, objects, relationships, etc. in object-oriented design are discussed in the sections ahead.

With a function-oriented design approach, the software units can be designed more simply, and this makes their implementation easier and less expensive. As every design unit is well defined, modification, deletion, and altering of the design units are faster. Functional design is supported by many programming languages. This approach has been followed informally since the beginning of the programming language. Some of the commonly used tools for function-oriented design are:

a) Entity-Relationship model: this model is used in the database to represent the real-world entity and relationship between them.

b) A decision tree is used for representing the actions and conditions in a software design using a graphical method. Decision tables represent the action and the related conditions in a tabular form.

c) DFDs are used for showing the flow of data graphically.

d) Structure charts are used to represent the system hierarchically. They represent every sub-system as a black box. The meaning of the black box is hiding the inner details and showing only what function it does.

A functional view of Compiler using DFD is as shown in Figure 7.2

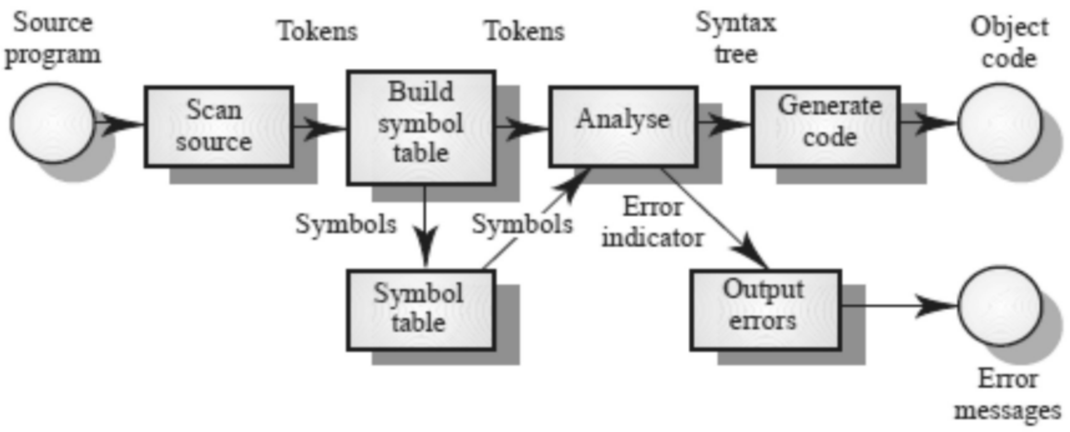


Figure 7.2 Functional view of a compiler

In the DFD, a rounded rectangle show function or transform, a rectangle displays data store. The circle is used for user interactions with the system arrows showing the direction of data flow or to link data flows with the help of a suitable keyword. Here in the above example, for the operation of the compiler, the scanning of a source program, parsing or building of a symbol table, error detection/analysis, and code generation are the steps that are functions/tasks. At the end of each task, the generated output such as symbols, errors is shown using proper links and keywords. The input to a compiler is a source program (in C, Java, etc) and output is an object program (executable file) and these two items are used for user interactions. These details are shown using circles.

7.6 DFDs and Structure Charts

Hierarchical Organization of Data Flow Diagrams

Any real-life situation with even moderate complexity will have many processes, data flows, and data stores. It is not desirable to show all of them in one DFD. Instead, for better comprehension, we normally organize them in more than one DFDs and hierarchically arrange them:

a) 0-level DFD or Context Diagram

A 0-level or context diagram normally has only one process (or referred to as one bubble) for the main task of the system and its relationship with external entities. In this type of DFD, a single bubble is used to represent the entire system along with input and output. A context diagram may also show the major data stores used in the system. Fig 7.3 shows the context diagram for the flight reservation.

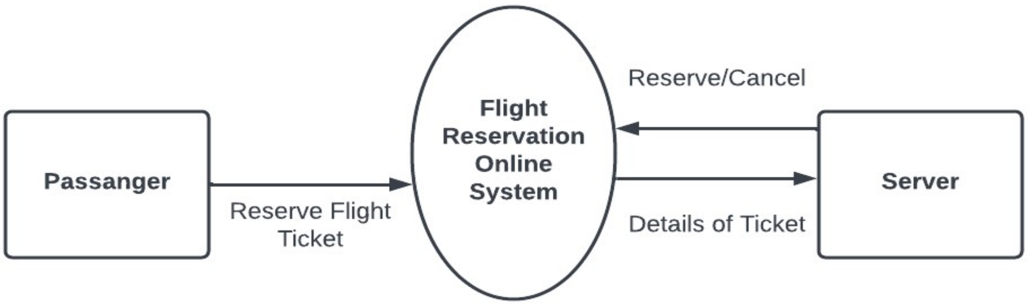


Fig. 7.3 Level-0 or Context diagram for the flight reservation

Here, the passenger is an external entity. The details of the reservation process are not shown here. The reservation process is depicted only as one process of the system.

b) 1-Level DFD: 1-level DFD depicts more than one process/bubble. The major process is highlighted and further, it is divided into subprocesses. As shown in Figure 7.4, a single reservation process is divided into subprocess named A1, A2, A3, and A4. Each subprocess is displaying relevant information such as passenger information, date of booking, transaction details, booking records, etc.

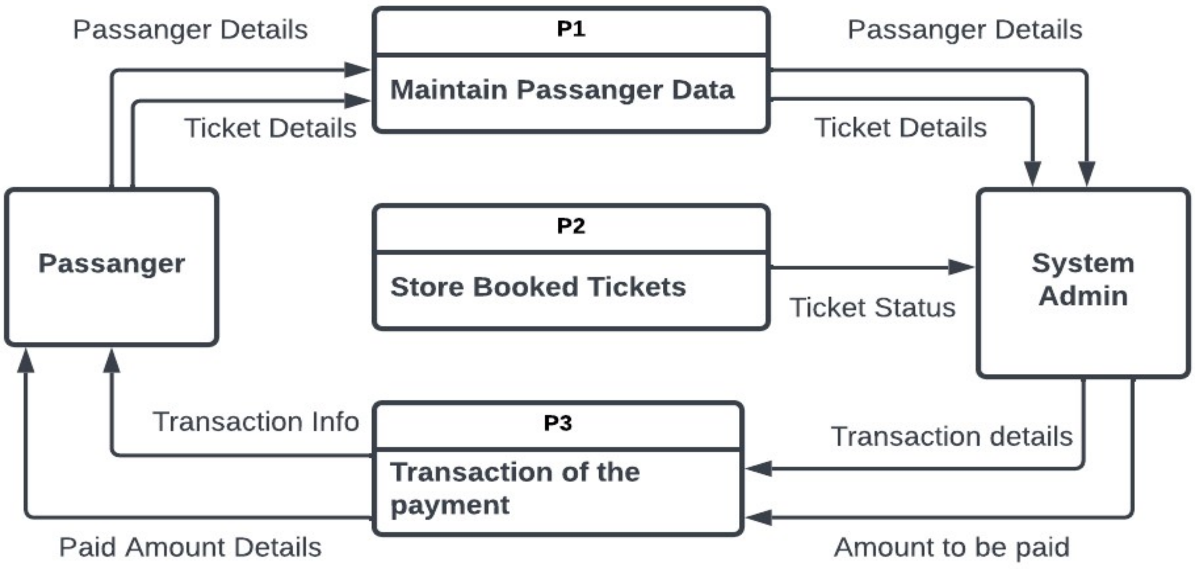


Fig.7.4 Level-1 DFD for the flight reservation

b) 2-level DFD

In Level 2 DFD, the data flow is also shown along with the processes. The reservation system DFD in level 2 is as shown in Figure 7.5. This DFD is based on DFD level 1 and level 0. In this diagram, the most important processes are managing customer information and booking record. The next priority goes to the processes for maintaining dates of bookings and updating the status of reserved tickets. Finally, monitoring transaction is yet another important process.

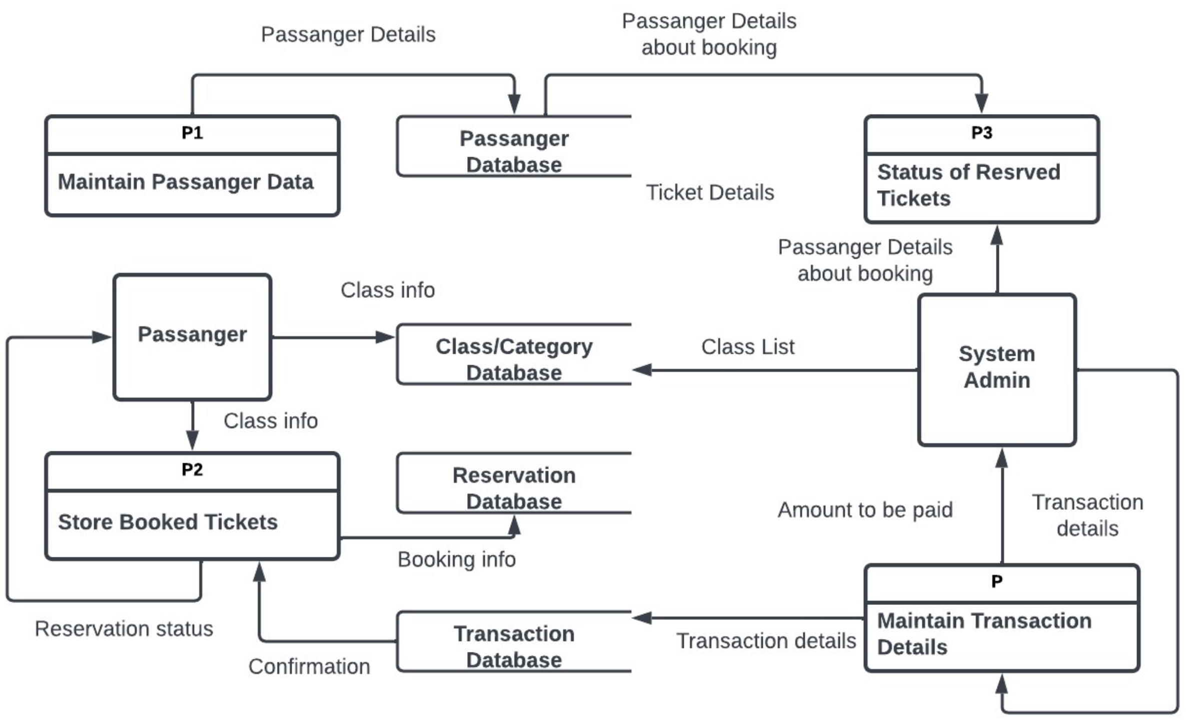


Figure 7.5 Level-2 DFD for the flight reservation

Weaknesses of Data Flow Diagrams

1. They lack precise meaning. Whereas their syntax, i.e., the way of composing the processes, arrows, and boxes, is defined precisely, their semantics is not.
2. They do not define the control aspects. For example, if a particular process will be executed only upon satisfaction of a condition, it cannot be depicted on the diagram.
3. One cannot test whether the specifications reflect a user's expectations. Thus, a traditional data flow diagram is a semi-formal notation.

Structure chart is yet another tool used in the function-oriented design, where the entire functional system is divided into lowest subunits and each unit is defined with input and output functions. Structure charts are derived from the DFDs, and they represent the system in a detailed manner.

The architecture of the software can be represented using a structure chart. The different modules in the system and the arguments or parameters passed to modules are represented using a structure chart. Once the structure charts are finalized, then they can be easily implemented using any programming language. The structure charts do not much give importance to how the functions are completed. Structure charts are different from flowcharts as: structure charts represent modules and data interchange among modules, this is difficult to do in flowcharts. Also, in the flowchart the functions are represented sequentially, this is suppressed in structural charts. The symbols in structure charts include:

Modules are represented by rectangular boxes. Arrows are used to pass control from one module to another with a given direction. Arrows are named with data to show the flow of data. Library modules are represented by a rectangle with double edges. The selection command is by using a diamond and loops are represented using the repetition around the control flow.

Conversion of DFD to Structure Charts: After expressing the design using DFD, the design can be converted into structure charts using three steps:

a) Divide the system into units that can be tracked.
b) Transform each unit into the form of a chart
c) Link the separated units to set up a complete system.

Consider the DFD for the reservation system as depicted in Figure 7.3.

The transaction involved here is check availability, book a flight ticket, or cancel a flight This DFD can be shown by a structure chart as in Figure 7.6.

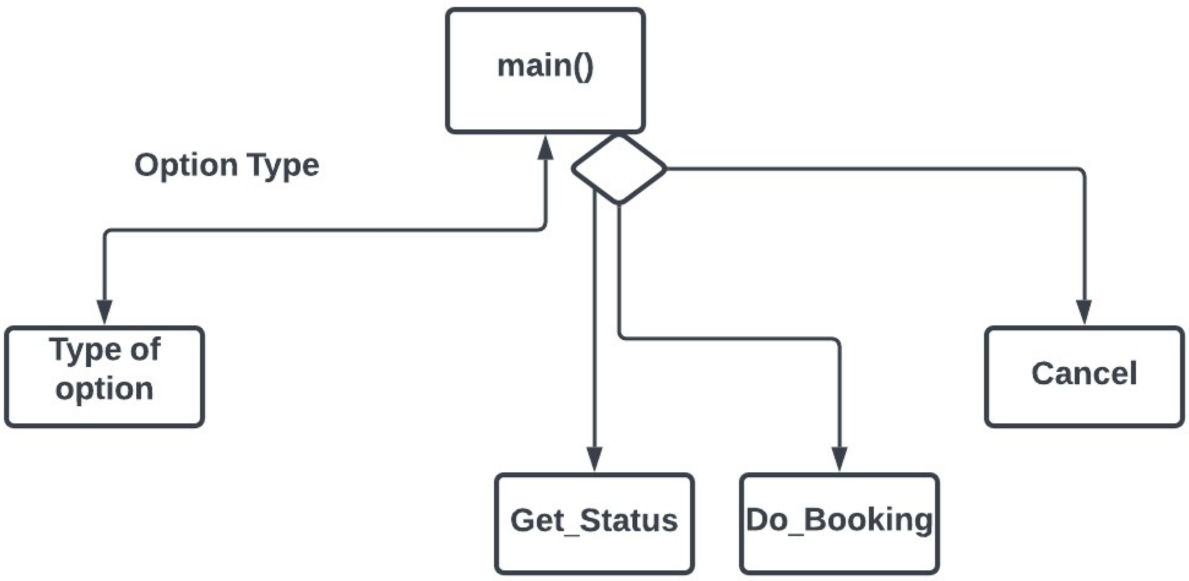


Figure 7.6 Structure Chart for the DFD

In the structure charts, the main () function is calling the remaining modules using the data Option Type. Based on the choice of the user, the modules, namely option Get-Status (), Do_Booking (), and Cancel () are called by the main (). In the next level, these transactions are further explored in a detailed manner.

7.7 Cohesion

Cohesion is a measure of the functional strength of a module. It is a measure of the degree to which the elements of a module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Cohesion is the internal glue that keeps the module together. A good software design will have high cohesion. The different classes of cohesion that a module may possess are depicted in Figure 7.7 below.

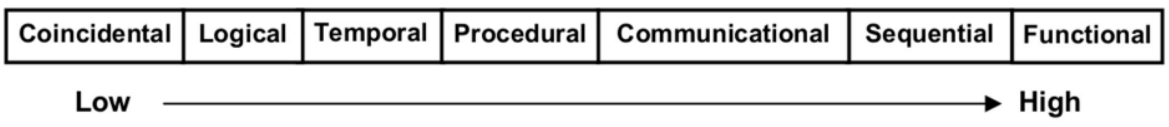


Figure 7.7 Classification of Cohesion

1) Coincidental Cohesion

A module is said to have coincidental cohesion if it performs a set of tasks that relate to each other very loosely, if at all. In this case, the module contains a random collection of functions. The functions have likely been put in the module out of pure coincidence without any thought or design. 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

2) Logical Cohesion

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

3) Temporal Cohesion

Temporal cohesion means all the functions of a module are executed in the same period.

Example: Set of functions for start, shutdown, sleep, initialize of any process in a computer

4) 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

5) Communicational 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

6) Sequential Cohesion

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

It occurs naturally in functional programming languages.

7) 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 seat_allotment module in the Airline_Ticket_reservation system.

Functional Cohesion is the most desirable

7.8 Coupling

Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. If two modules interchange large amounts of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity. The interface complexity is determined by the number of types of parameters that are interchanged while invoking the functions of the module. Figure 7.8 shows the different classes of coupling.

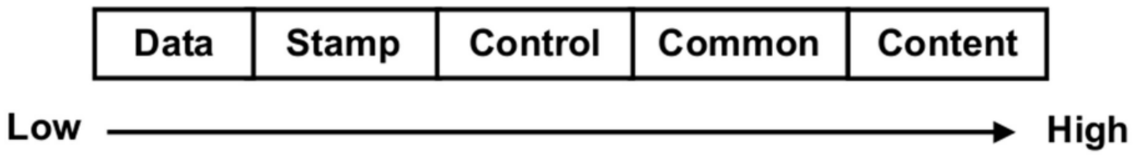


Figure 7.8 Classification of Coupling

1) Data Coupling

Two modules are data-coupled if they communicate through a parameter. An example is an elementary data item (like integer, float, character) passed as a parameter between two modules. This data item should be problem-related and not used for control purposes. In the following example, the data bank balance is used as a common parameter.

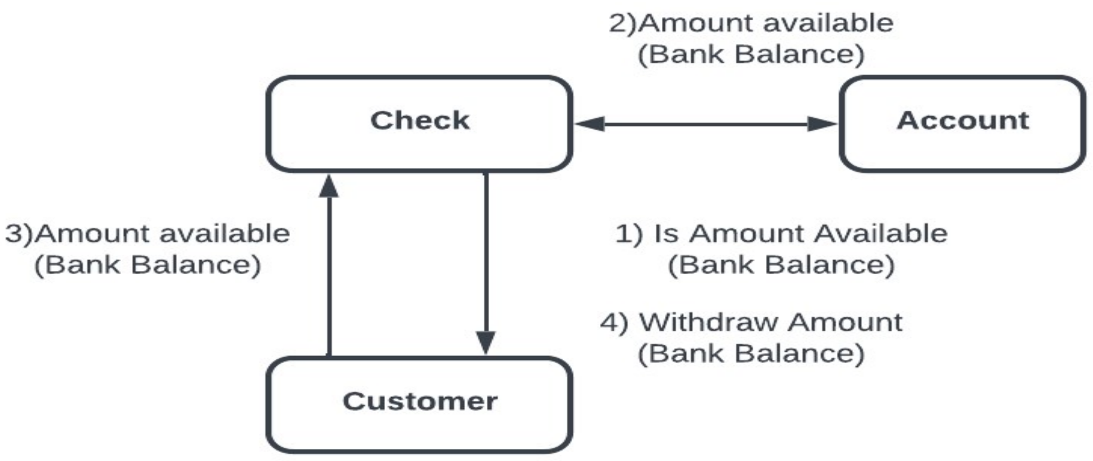


Figure 7.9 Data Coupling

2) Stamp Coupling

Two modules are stamp coupled if they communicate using a composite data item such as a structure in C.

3) 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()}
```

4) Common Coupling

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

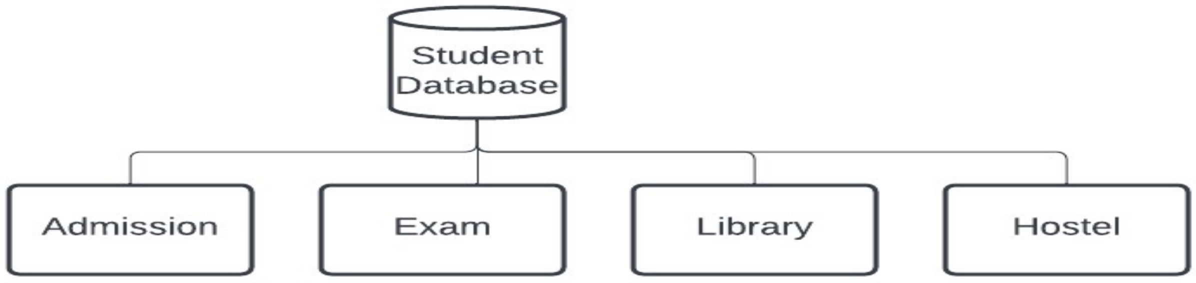


Figure 7.10 Common Coupling

5) Content Coupling

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

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

7.9 Functional Independence

A module with high cohesion and low coupling is said to be functionally independent of other modules. When a cohesive module performs a single task or function, it is known as functionally independent. A functionally independent module has minimal interaction with other modules.

Functional Independence is required for a good design due to the following reasons:

1. Error Isolation: If a module is functionally independent, its degree of interaction with the other modules is less. Therefore, any error existing in a module would not directly affect the other modules.
2. Scope of Reuse: Each module does some well-defined and precise function, and the interaction of the module with the other modules is simple and minimal. Therefore, a cohesive module can be easily taken out and reused in a different program.
3. Understandability: The complexity of the design is reduced because different modules can be understood in isolation as modules are more or less independent of each other. independent of each other.

7.10 Modularity

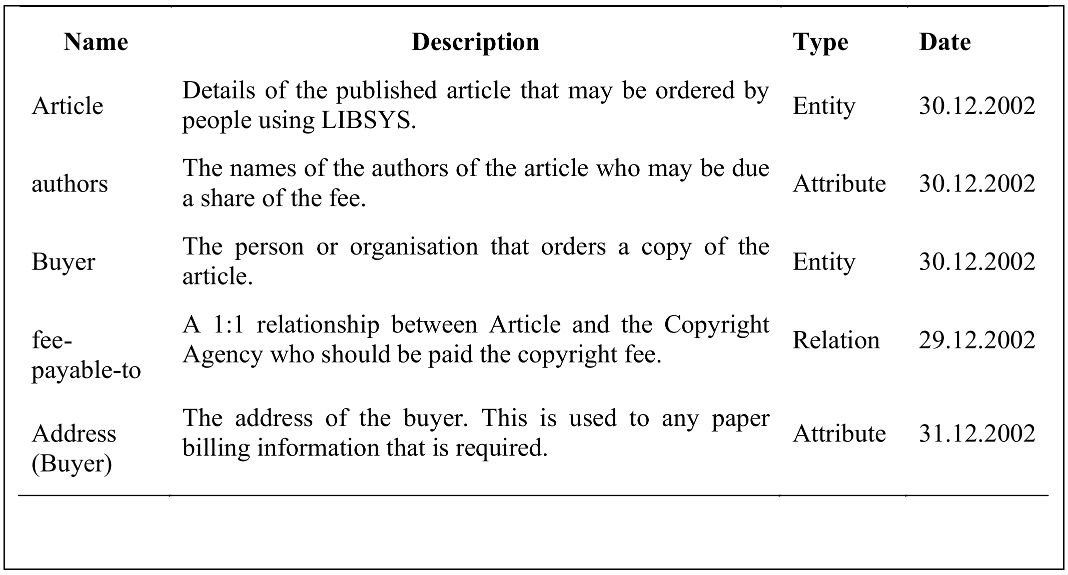
A good software design implies a clean decomposition of the problem into modules and the neat arrangement of these modules in a hierarchy. These modules should be mostly independent of each other. It should be easy to understand each module separately, hence reducing the complexity.

The primary characteristics of neat module decomposition are high cohesion and low coupling. A module having high cohesion and low coupling is said to be functionally independent of other modules. There are many advantages of modular design in software engineering. Some of these are:

1. Easy to understand the system.
2. System maintenance is easy.
3. A module can be used many times as their requirements. No need to write it again and again.

7.11 Detailed Design

In a detailed design, a short description of each function is presented. The different operations, input, and output are shown in this step. The data dictionary is a tool to describe the detailed design. The output of this step is used in process design language, commonly known as PDL. The PDL is written in structured English and can be converted to a high-level programming language. One example of a data dictionary used in the detailed design for the library management software design is shown in Figure 7.11.



Name	Description	Type	Date
Article	Details of the published article that may be ordered by people using LIBSYS.	Entity	30.12.2002
authors	The names of the authors of the article who may be due a share of the fee.	Attribute	30.12.2002
Buyer	The person or organisation that orders a copy of the article.	Entity	30.12.2002
fee-payable-to	A 1:1 relationship between Article and the Copyright Agency who should be paid the copyright fee.	Relation	29.12.2002
Address (Buyer)	The address of the buyer. This is used to any paper billing information that is required.	Attribute	31.12.2002

Figure 7.11 Data Dictionary for LIBSYS software: Library Management

7.12 Multiple Choice Questions (MCQs)

1) The dependency between different modules of a software design is given by:
    a) Cohesion
    b) Parameter Passing
    c) Coupling
    d) Return Value
2) An error handling function that gets executed at the same time in more than one program has cohesion known as:
    a) Functional Cohesion
    b) Temporal Cohesion
    c) Sequential Cohesion
    d) Coincidental Cohesion
3) The rectangles in DFDs are used for:
    a) Function
    b) Data flow
    c) Data Store
    d) Process
4) The 2-Level DFD in functional design is also known as:
    a) Context diagram
    b) Second rank diagram
    c) 0-Level DFD
    d) All the above
5) DFDs can be used to create:
    a) Data dictionary
    b) Decision trees
    c) Structure charts
    d) All the above
6) The number of levels in DFD is:
    a) 1
    b) 2
    c) 3
    d) 4

7) When the software design is to start from nothing then the suitable design strategy is:
   a) Top-Down software design
   b) Bottom-up software design
   c) Interface design
   d) Design Patterns
8) The false statement about modularity in software design is
   a) Dependency in modules is preferred
   b) Modules can define the functionality in a better way
   c) Modules are easy to understand
   d) Debugging a module is easier than the whole program
9) When one module is returning a value and directs the flow of execution in another module, then the coupling used is:
   a) Data coupling
   b) Control coupling
   c) Strong coupling
   d) Stamp coupling
10) Consider a function PRINT (), used in more than one module to print the data. Then the type of cohesion referred to in this design is:
   a) Sequential cohesion
   b) Functional cohesion
   c) Print cohesion
   d) Communicational cohesion

7.13 REVIEW QUESTIONS

1) What are the characteristics of a good design? Explain with an example of library management software design.
2) Draw a DFD for ATM design and explain the different symbols.
3) Explain 0-level, 1-level, and 2-level DFDs with an example of student admission in GEU.
4) Draw the DFD and structure chart for the online movie booking software system.
5) Compare DFDs and structure charts
6) What are cohesion and coupling? Explain with a suitable example

7) What are the types of cohesion? Explain the application of each type with a suitable example.
8) Write any C programming code snippet showing communication coupling.

References

The content has been composed referring to various online and offline resources. Major sources of reference are listed below. Copyright exists with the respective organizations. Contents have been adapted for educational purposes only. No commercial benefits are derived.

- Sommerville, I. (2016) Software Engineering. 10th Edition, Pearson Education Limited, Boston.
- Pressman, R.S. (2010) Software Engineering: A Practitioner's Approach. 7th Edition, McGraw Hill, New York.

7.14 MCQ solution keys

1:c	2: b	3: d	4: b	5: c	6:c	7: b	8: a	9: b	10: b