Module 7

Software Engineering Issues

Lesson 36

Software Design – Part 1

Specific Instructional Objectives

At the end of this lesson, the student would be able to:

- Identify the software design activities
- State the desirable characteristics of a good software design
- Understand cohesion and coupling
- Explain the importance of functional independence in software design
- State the features of a function-oriented design approach
- State the features of an object-oriented design approach
- Differentiate between function-oriented and object-oriented design approach
- Identify the activities carried out during the structured analysis phase
- Explain the Data Flow Diagram and its importance in software design
- Explain the Data Dictionary and its importance
- Identify whether a DFD is balanced
- Draw the context diagram of any given problem
- Draw the DFD model of any given problem
- Develop the data dictionary for any given problem
- Identify common errors that can occur while constructing a DFD model
- Identify the shortcomings of a DFD model
- Differentiate between a structure chart and a flow chart
- Identify the activities carried out during transform analysis with examples
- Explain what is meant by transaction analysis

1. Introduction

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. A good software design is seldom arrived by using a single step procedure, but requires several iterations through a series of steps. Design activities can be broadly classified into two important parts:

- Preliminary (or high-level) design and
- Detailed design

High-level design means identification of different modules and the control relationships among them and the definition of the interfaces among these modules. The outcome of high-level design is called the program structure or software architecture. During detailed design, the data structure and the algorithms of the different modules are designed. The outcome of the detailed design stage is usually known as the module-specification document.

1.1. Characteristics of a Good Software Design

However, most researchers and software engineers agree on a few desirable characteristics that every good software design for general application must possess. They are listed below:

Correctness: A good design should correctly implement all the functionalities identified in the SRS document.

Understandability: A good design is easily understandable.

Efficiency: It should be efficient.

Maintainability: It should be easily amenable to change.

1.2. Current Design Approaches

Most researchers and engineers agree that a good software design implies clean decomposition of the problem into modules, and the neat arrangement of these modules in a hierarchy. The primary characteristics of neat module decomposition are high cohesion and low coupling.

1.2.1. Cohesion

Most researchers and engineers agree that a good software design implies clean decomposition of the problem into modules, and the neat arrangement of these modules in a hierarchy. The primary characteristics of neat module decomposition are high cohesion and low coupling.

Cohesion is a measure of functional strength of a module. A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. The different classes of cohesion that a module may possess are depicted in fig. 36.1.

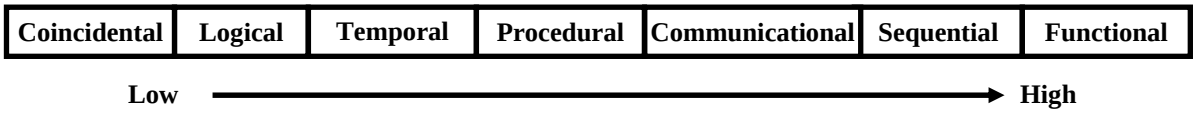


Fig. 36.1 Classification of Cohesion

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. It is likely that the functions have been put in the module out of pure coincidence without any thought or design.

Logical cohesion: A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc. An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

Temporal cohesion: When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal

cohesion. The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion.

Procedural cohesion: A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which a certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.

Communicational cohesion: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack.

Sequential cohesion: A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next.

Functional cohesion: Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' pay-roll displays functional cohesion. Suppose a module displays functional cohesion, and we are asked to describe what the module does, then we would be able to describe it using a single sentence.

1.2.2. Coupling

Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module having high cohesion and low coupling is said to be functionally independent of other 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 basically determined by the number of types of parameters that are interchanged while invoking the functions of the module. Even if no techniques to precisely and quantitatively estimate the coupling between two modules exist today, classification of the different types of coupling will help to quantitatively estimate the degree of coupling between two modules. Five types of coupling can occur between any two modules as shown in fig. 36.2.

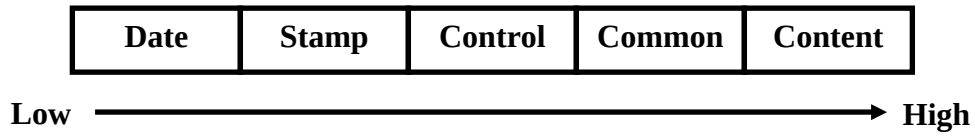


Fig. 36.2 Classification of coupling

Stamp Coupling: Two modules are stamped coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling: Control coupling exists between two couples, if data from one module is used to direct the order of instructions execution in another. An example of control coupling is a flag set in one module and tested in another module.

Common coupling: Two modules are common coupled, if they share some global data items.

Content coupling: Content coupling exists between two modules, if their code is shared, e.g. a branch from one module into another module.

1.2.3. Functional Independence

A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

Functional independence is a key to any good design primarily due to the following reasons:

Error isolation: Functional independence reduces error propagation. The reason behind this is that 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 effect the other modules**.**

Scope of reuse: Reuse of a module becomes possible- because 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**.**

Understandability: Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.

1.2.4. Function-Oriented Design Approach

The following are the salient features of a typical function-oriented design approach:

1. A system is viewed as something that performs a set of functions. Starting at this high-level view of the system, each function is successively refined into more detailed functions. For example, consider a function create-new-library member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. This function may consist of the following sub-functions:
   - assign-membership-number
   - create-member-record
   - print-bill

   Each of these sub-functions may be split into more detailed sub-functions and so on.
2. The system state is centralized and shared among different functions, e.g. data such as member-records is available for reference and updating to several functions such as:
   - create-new-member
   - delete-member
   - update-member-record

1.2.5. Object-Oriented Design Approach

In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information. For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. In fact, the functions defined for one object cannot refer or change data of other objects. Objects have their own internal data which define their state. Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from super class. Conceptually, objects communicate by message passing.

1.2.6. Function-Oriented Vs. Object-Oriented Design

The following are some of the important differences between function-oriented and object-oriented design.

- Unlike function-oriented design methods, in OOD, the basic abstraction are not real-world functions such as sort, display, track, etc, but real-world entities such as employee, picture, machine, radar system, etc. For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee-record, get-employee-address, etc. but by designing objects such as employees, departments, etc.
- In object-oriented design, software is not developed by designing functions such as update-employee-record, get-employee-address, etc., but by designing objects such as employee, department, etc.
- In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system. For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc. are usually implemented as global data in a traditional programming system; whereas in an object-oriented system these data are distributed among different employee objects of the system. Objects communicate by passing messages. Therefore, one object may discover the state information of another object by interrogating it. Of course, somewhere or the other the real-world functions must be implemented.
- Function-oriented techniques such as SA/SD group functions together if, as a group, they constitute a higher-level function. On the other hand, object-oriented techniques group functions together on the basis of the data they operate on.

To illustrate the differences between the object-oriented and the function-oriented design approaches, an example can be considered.

Example: Fire-Alarm System

The owner of a large multi-storied building wants to have a computerized fire alarm system for his building. Smoke detectors and fire alarms would be placed in each room of the building. The fire alarm system would monitor the status of these smoke detectors. Whenever a fire condition is reported by any of the smoke detectors, the fire alarm system should determine the location at which the fire condition is reported by any of the smoke detectors. The fire alarm system should determine the location at which the fire condition has occurred and then sound the alarms only in the neighboring locations. The fire alarm system should also flash an alarm message on the computer consol. Fire fighting personnel man the console round the clock. After a fire condition has been successfully handled, the fire alarm system should support resetting the alarms by the fire fighting personnel.

Function-Oriented Approach:

```
/* Global data (system state ) accessible by various functions */

BOOL detector_status[MAX_ROOMS];
int detector_locs[MAX_ROOMS];
BOOL alarm_status[MAX_ROOMS];/* alarm activated when status is set */
int alarm_locs[MAX_ROOMS];   /* room number where alarm is located */
int neighbor-alarm[MAX_ROOMS][10];
/* each detector has at most 10 neighboring locations */

The functions which operate on the system state are:
interrogate_detectors();
get_detector_location();
determine_neighbor();
ring_alarm();
reset_alarm();
report_fire_location();
```

Object-Oriented Approach:

```
class detector
attributes: status, location, neighbors
operations: create, sense-status, get-location, find-neighbors

class alarm
attributes: location, status
operations: create, ring-alarm, get_location, reset-alarm
```

In the object oriented program, an appropriate number of instances of the class detector and alarm should be created. If the function-oriented and the object-oriented programs are examined, then it is seen that in the function-oriented program the system state is centralized and several functions on this central data is defined. In case of the object-oriented program, the state information is distributed among various objects.

It is not necessary that an object-oriented design be implemented by using an object-oriented language only. However, an object-oriented language such as C++, supports the definition of all the basic mechanisms of class, inheritance, objects, methods, etc., and also supports all key object-oriented concepts that we have just discussed. Thus, an object-oriented language facilitates the implementation of an OOD. However, an OOD can as well be implemented using a conventional procedural language – though it may require more effort to implement an OOD using a procedural language as compared to the effort required for implementing the same design using an object-oriented language.

Even though object-oriented and function-oriented approaches are remarkably different approaches to software design, they do not replace each other but complement each other in some sense. For example, usually one applies the top-down function oriented techniques to design the internal methods of a class, once the classes are identified. In this case, though outwardly the system appears to have been developed in an object-oriented fashion, inside each class there may be a small hierarchy of functions designed in a top-down manner.

2. Function-Oriented Software Design

Function-oriented design techniques view a system as a black-box that performs a set of high-level functions. During the design process, these high-level functions are successively decomposed into more detailed functions and finally the different identified functions are mapped to modules. The term *top-down decomposition* is often used to denote such successive decompositions of a set of high-level functions into more detailed functions.

2.1. Structured Analysis

Structured analysis is used to carry out the top-down decomposition of a set of high-level functions depicted in the problem description and to represent them graphically. During structured analysis, functional decomposition of the system is achieved. That is, each function that the system performs is analysed and hierarchically decomposed into more detailed functions. Structured analysis technique is based on the following essential underlying principles:

- Top-down decomposition approach.
- Divide and conquer principle. Each function is decomposed independently.
- Graphical representation of the analysis results using Data Flow Diagrams (DFDs).

2.2. Data Flow Diagrams

The DFD (also known as a bubble chart) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system. A DFD model uses a very limited number of primitive symbols (as shown in fig. 36.3) to represent the functions performed by a system and the data flow among these functions.

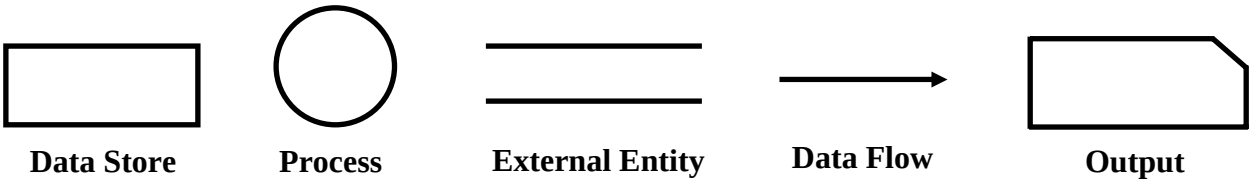


Fig. 36.3 Symbols used for designing DFDs

The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism – it is simple to understand and use. Starting with a set of high-level functions that a system performs, a DFD model hierarchically represents various sub-functions. In fact, any hierarchical model is simple to understand. The human mind is such that it can easily understand any hierarchical model of a system – because in a hierarchical model, starting with a very simple and abstract model of a system, different details of the system are slowly introduced through different hierarchies. The data flow diagramming technique also follows a very simple set of intuitive concepts and rules. DFD is an elegant modeling technique that turns out to be useful not only to represent the results of structured analysis of a software problem but also for several other applications such as showing the flow of documents or items in an organization.

2.2.1. Data Dictionary

A data dictionary lists all data items appearing in the DFD model of a system. The data items listed include all data flows and the contents of all data stores appearing on the DFDs in the DFD model of a system.

A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items. For example, a data dictionary entry may represent that the data *grossPay* consists of the components *regularPay* and *overtimePay*.

grossPay = regularPay + overtimePay

For the smallest units of data items, the data dictionary lists their name and their type.

A data dictionary plays a very important role in any software development process because of the following reasons:

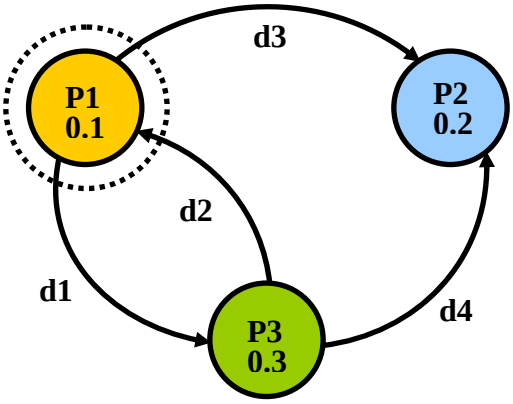
- A data dictionary provides a standard terminology for all relevant data for use by engineers working in a project. A consistent vocabulary for data items is very important, since in large projects different engineers of the project have a tendency to use different terms to refer to the same data, which unnecessarily causes confusion.
- The data dictionary provides the analyst with a means to determine the definition of different data structures in terms of their component elements.

2.3. DFD : Levels and Model

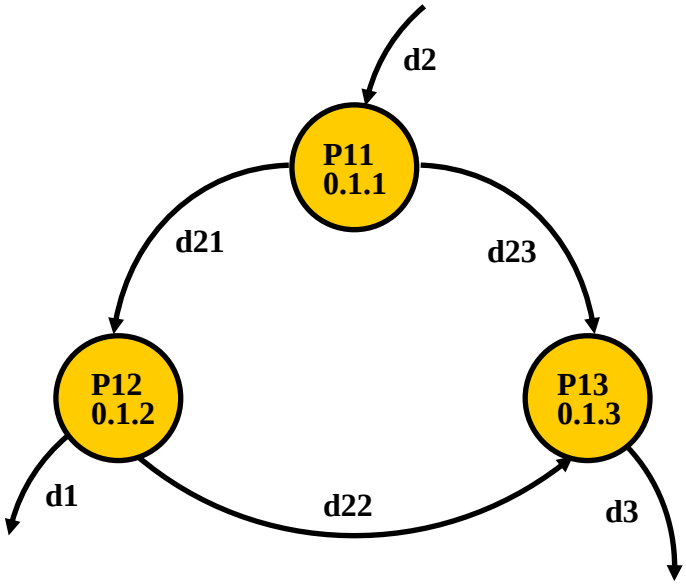
The DFD model of a system typically consists of several DFDs, viz., level 0 DFD, level 1 DFD, level 2 DFDs, etc. A single data dictionary should capture all the data appearing in all the DFDs constituting the DFD model of a system.

2.3.1. Balancing DFDs

The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD. The concept of balancing a DFD has been illustrated in fig. 36.4. In the level 1 of the DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble P1. In the next level, bubble 0.1 is decomposed. The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in.



(a) Level 1 DFD



(b) Level 2 DFD

Fig. 36.4 An example showing balanced decomposition

2.3.2. Context Diagram

The context diagram is the most abstract data flow representation of a system. It represents the entire system as a single bubble. This bubble is labeled according to the main function of the system. The various external entities with which the system interacts and the data flow occurring between the system and the external entities are also represented. The data input to the system and the data output from the system are represented as incoming and outgoing arrows. These data flow arrows should be annotated with the corresponding data names. The name context diagram is well justified because it represents the context in which the system is to exist, i.e. the external entities who would interact with the system and the specific data items they would be supplying the system and the data items they would be receiving from the system. The context diagram is also called the level 0 DFD.

To develop the context diagram of the system, we have to analyse the SRS document to identify the different types of users who would be using the system and the kinds of data they would be inputting to the system and the data they would be receiving from the system. Here, the term “users of the system” also includes the external systems which supply data to or receive data from the system.

The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). This is in contrast with the bubbles in all other levels which are annotated with verbs. This is expected since the purpose of the context diagram is to capture the context of the system rather than its functionality.

Example 1: RMS Calculating Software

A software system called RMS calculating software would read three integral numbers from the user in the range of -1000 and +1000 and then determine the root mean square (rms) of the three input numbers and display it. In this example, the context diagram (fig. 36.5) is simple to draw. The system accepts three integers from the user and returns the result to him.

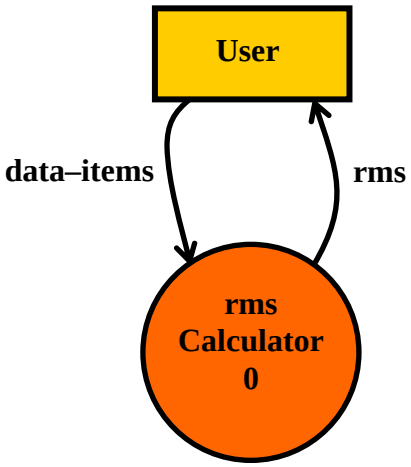


Fig. 36.5 Context Diagram

Example 2: Tic-Tac-Toe Computer Game

Tic-tac-toe is a computer game in which a human player and the computer make alternative moves on a 3 × 3 square. A move consists of marking previously unmarked square. The player, who is first to place three consecutive marks along a straight line (i.e. along a row, column, or diagonal) on the square, wins. As soon as either of the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, nor all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game. The context diagram of this problem is shown in fig. 36.6.

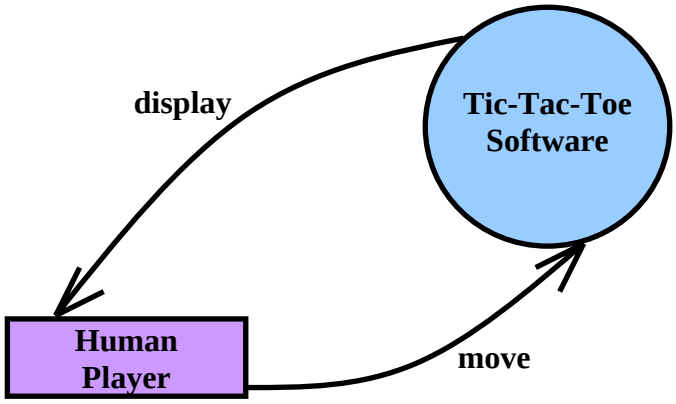


Fig. 36.6 Context diagram for tic-tac-toe computer game

2.3.3. Developing the DFD Model

A DFD model of a system graphically depicts the transformation of the data input to the system to the final result through a hierarchy of levels. A DFD starts with the most abstract definition of the system (lowest level) and at each higher level DFD, more details are successively introduced. To develop a higher-level DFD model, processes are decomposed into their sub-processes and the data flow among these sub-processes is identified.

To develop the data flow model of a system, first the most abstract representation of the problem is to be worked out. The most abstract representation of the problem is also called the context diagram. After, developing the context diagram, the higher-level DFDs have to be developed.

Context Diagram

Level 1 DFD: To develop the level 1 DFD, examine the high-level functional requirements. If there are between 3 to 7 high-level functional requirements, then these can be directly represented as bubbles in the level 1 DFD. We can then examine the input data to these functions, the data output by these functions, and represent them appropriately in the diagram.

If a system has more than 7 high-level functional requirements, then some of the related requirements have to be combined and represented in the form of a bubble in the level 1 DFD. Such a bubble can be split in the lower DFD levels. If a system has less than three high-level functional requirements, then some of them need to be split into their sub-functions so that we have roughly about 5 to 7 bubbles on the diagram.

Decomposition: Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into sub-functions at the successive levels of the DFD. Decomposition of a bubble is also known as factoring or exploding a bubble. Each bubble at any level of DFD is usually decomposed to anything between 3 to 7 bubbles. Too few bubbles at any level make that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes redundant. Also, too many bubbles, i.e. more than 7 bubbles at any level of a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm.

Numbering the Bubbles: It is necessary to number the different bubbles occurring in the DFD. These numbers help in uniquely identifying any bubble in the DFD from its bubble number. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc. When a bubble numbered x is decomposed, its children bubble are numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble, we can unambiguously determine its level, its ancestors and its successors.

Example: Supermarket Prize Scheme

A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 carat gold coin to every customer whose purchase exceeds Rs.10,000. The entries against the CN are the reset on the day of every year after the prize winners' lists are generated.

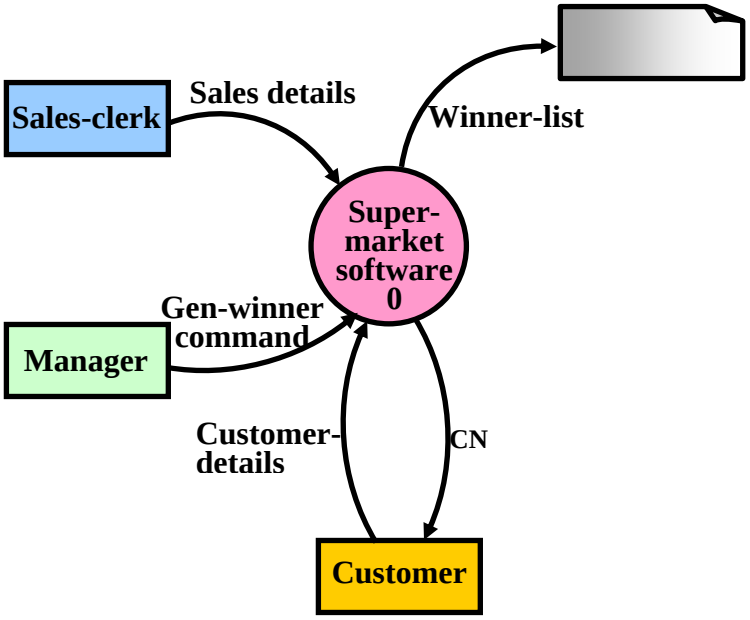


Fig. 36.7 Context diagram for supermarket problem

The context diagram for this problem is shown in fig. 36.7, the level 1 DFD in fig. 36.8, and the level 2 DFD in fig. 36.9.

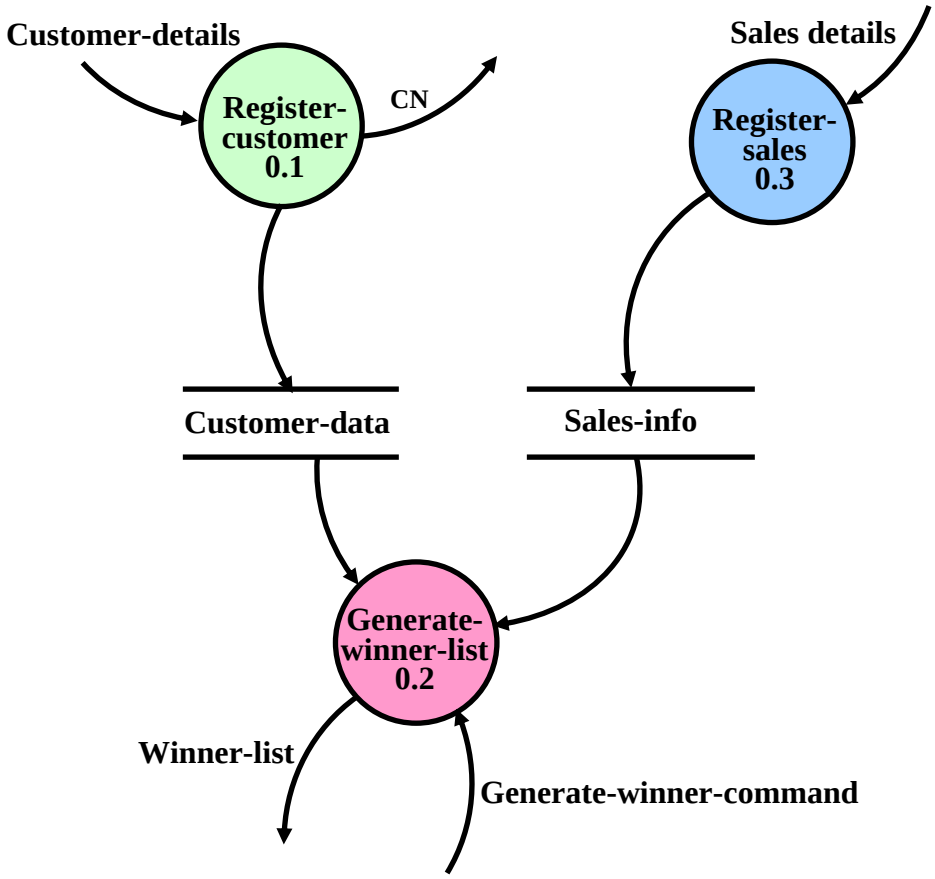


Fig. 36.8 Level 1 diagram for supermarket problem

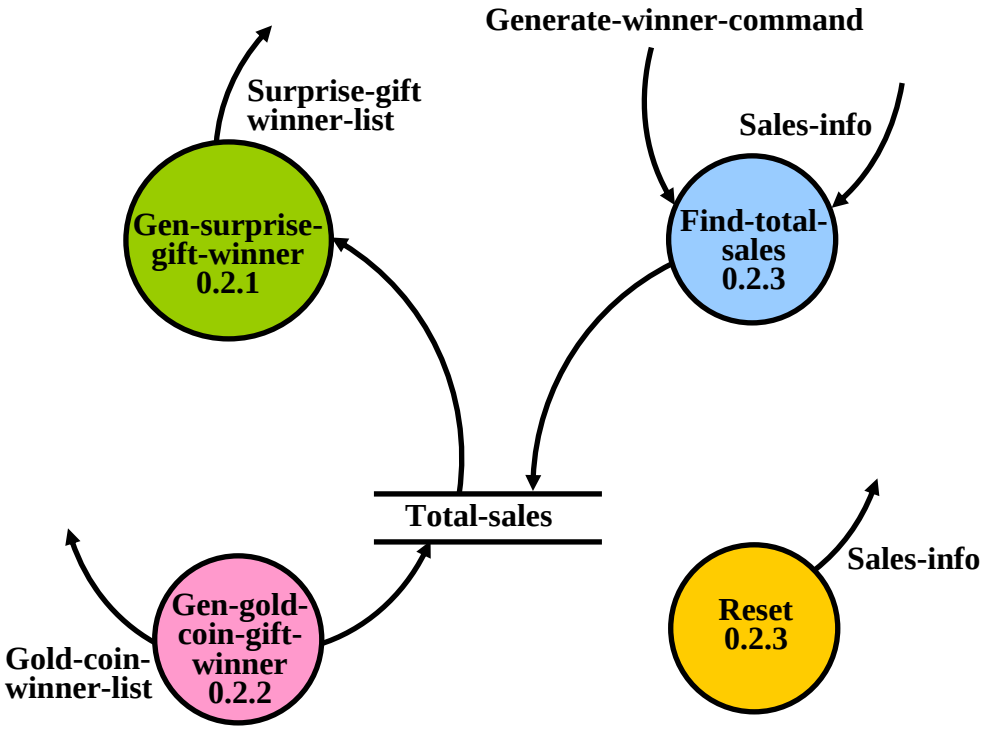


Fig. 36.9 Level 2 diagram for supermarket problem

Data Dictionary for the DFD Model

address: name + house# + street# + city + pin
sales-details: {item + amount}* + CN
CN: integer
customer-data: {address + CN}*
sales-info: {sales-details}*
winner-list: surprise-gift-winner-list + gold-coin-winner-list
surprise-gift-winner-list: {address + CN}*
gold-coin-winner-list: {address + CN}*
gen-winner-command: command
total-sales: {CN + integer}*

2.3.4. Common Errors in Constructing DFD Model

Although DFDs are simple to understand and draw, students and practitioners alike encounter similar types of problems while modelling software problems using DFDs. While learning from experience is a powerful thing, it is an expensive pedagogical technique in the business world. It is therefore helpful to understand the different types of mistakes that users usually make while constructing the DFD model of systems.

- Many beginners commit the mistake of drawing more than one bubble in the context diagram. A context diagram should depict the system as a single bubble.
- Many beginners have external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear at other levels of the DFD.
- It is a common oversight to have either too less or too many bubbles in a DFD. Only 3 to 7 bubbles per diagram should be allowed, i.e. each bubble should be decomposed to between 3 and 7 bubbles.
- Many beginners leave different levels of DFD *unbalanced*.
- A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD. It is important to realize that a DFD is the data flow representation of a system and it does not represent control information. The following examples represent some mistakes of this kind:
    - A book can be searched in the library catalogue by inputting its name. If the book is available in the library, then the details of the book are displayed. If the book is not listed in the catalogue, then an error message is generated. While generating the DFD model for this simple problem, many beginners commit the mistake of drawing an arrow (as shown in fig. 36.10) to indicate the error function is invoked after the search book. But, this is a control information and should not be shown on the DFD.

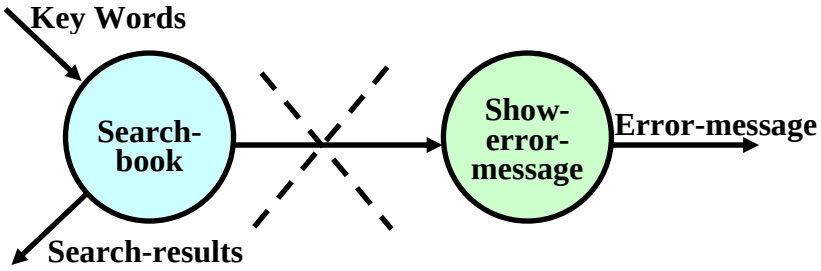


Fig. 36.10 To show control information on a DFD – A mistake

- Another error is trying to represent when or in what order different functions (processes) are invoked and the conditions under which different functions are invoked.
- If a bubble A invokes either the bubble B or the bubble C depending upon some conditions, we need only to represent the data that flows between bubbles A and B or bubbles A and C and not the conditions depending on which the two modules are invoked.

- A data store should be connected only to bubbles through data arrows. A data store cannot be connected to either another data store or to an external entity.
- All the functionalities of the system must be captured by the DFD model. No function of the system specified in its SRS document should be overlooked.
- Only those functions of the system specified in the SRS document should be represented, i.e. the designer should not assume functionality of the system not specified by the SRS document and then try to represent them in the DFD.
- Improper or unsatisfactory data dictionary.
- The data and function names must be intuitive. Some students and even practicing engineers use symbolic data names such a, b, c, etc. Such names hinder understanding the DFD model.

2.3.5. Shortcomings of a DFD Model

DFD models suffer from several shortcomings. The important shortcomings of the DFD models are the following:

- DFDs leave ample scope to be imprecise. In the DFD model, we judge the function performed by a bubble from its label. However, a short label may not capture the entire functionality of a bubble. For example, a bubble named find-book-position has only intuitive meaning and does not specify several things, e.g. what happens when some input information is missing or is incorrect. Further, the find-book-position bubble may not convey anything regarding what happens when the required book is missing.
- Control aspects are not defined by a DFD. For instance, the order in which inputs are consumed and outputs are produced by a bubble is not specified. A DFD model does not specify the order in which the different bubbles are executed. Representation of such aspects is very important for modeling real-time systems.

- The method of carrying out decomposition to arrive at the successive levels and the ultimate level to which decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst. Due to this reason, even for the same problem, several alternative DFD representations are possible. Further, many a times it is not possible to say which DFD representation is superior or preferable to another.
- The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its sub-functions and we have to use subjective judgment to carry out decomposition.

2.3.6. Extending DFD Technique To Real-Time Systems

The aim of structured design is to transform the results of the structured analysis (i.e. a DFD representation into a structure chart). A structure chart represents the software architecture, i.e. the various modules making up the system, the module dependency, and the parameters that are passed among the different modules. Since the main focus in a structure chart representation is on the module structure of software and the interaction between the different modules, the procedural aspects are not represented.

A real-time system is one where the functions must not only produce correct result but also should produce them by some pre-specified time. For real-time systems since reasoning about time is important to come up with a correct design, explicit representation of control and event flow aspects are essential. One of the widely accepted techniques for extending the DFD technique to real-time system analysis is the Ward and Mellor technique [1985]. In the Ward and Mellor notation, a type of process that handles only control flows is introduced. These processes representing control processing are denoted using dashed bubbles. Control flows are shown using dashed lines/arrows.

Unlike Ward and Mellor, Hatley and Pirbhai [1987] show the dashed and solid representations on separate diagrams. To be able to separate the data processing and the control processing aspects, a Control Flow Diagram (CFD) is defined. This reduces the complexity of the diagrams. In order to link the data processing and control processing diagrams, a notational reference (solid bar) to a control specification is used. The CSPEC describes the following:

- The effect of an external event or control signal
- The processes that are invoked as a consequence of an event

Control specifications represent the behaviour of the system in two different ways:

- It contains a state transition diagram (STD). The STD is a sequential specification of behaviour.
- It contains a program activation table (PAT). The PAT is a combinational specification of behaviour. PAT represents invocation sequence of bubbles in a DFD.

2.4. Structured Design

The aim of structured design is to transform the results of the structured analysis (i.e. a DFD representation into a structure chart). A structure chart represents the software architecture, i.e. the various modules making up the system, the module dependency, and the parameters that are passed among the different modules. Since the main focus in a structure chart representation is on the module structure of software and the interaction between the different modules, the procedural aspects are not represented.

2.4.1. Flow Chart Vs. Structure Chart

We are all familiar with the flow chart representation of a program. Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

- It is usually difficult to identify the different modules of the software from its flow chart representation.
- Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

2.4.2. Transformation of a DFD into a Structure Chart

Systematic techniques are available to transform the DFD representation of a problem into a module structure represented by a structure chart. Structured design provides two strategies:

- Transform Analysis
- Transaction Analysis

2.4.3. Transform Analysis

Transform analysis identifies the primary functional components (modules) and the high level inputs and outputs for these components. The first step in transform analysis is to divide the DFD into 3 types of parts:

- Input
- Logical processing
- Output

The input portion of the DFD includes processes that transform input data from physical (e.g. character from terminal) to logical forms (e.g. internal tables, lists, etc.). Each input portion is called an *afferent* branch.

The output portion of a DFD transforms output data from logical to physical form. Each output portion is called *efferent* branch. The remaining portion of a DFD is called *central transform*.

In the next step of transform analysis, the structure chart is derived by drawing one functional component for the central transform, and the afferent and efferent branches. These are drawn below a root module, which would invoke these modules.

Identifying the highest level input and output transforms requires experience and skill. One possible approach is to trace the inputs until a bubble is found whose output cannot be deduced from its inputs alone. Processes which validate input or add information to them are not central transforms. Processes which sort input or filter data from it are. The first level structure chart is produced by representing each input and output unit as boxes and each central transform as a single box.

In the third step of transform analysis, the structure chart is refined by adding sub-functions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, error-handling modules, initialization and termination process, identifying customer modules etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

Example: Structure chart for the RMS software

For this example, the context diagram was drawn earlier.
To draw the level 1 DFD (fig. 36.11), from a cursory analysis of the problem description, we can see that there are four basic functions that the system needs to perform – accept the input numbers from the user, validate the numbers, calculate the root mean square of the input numbers and, then display the result.

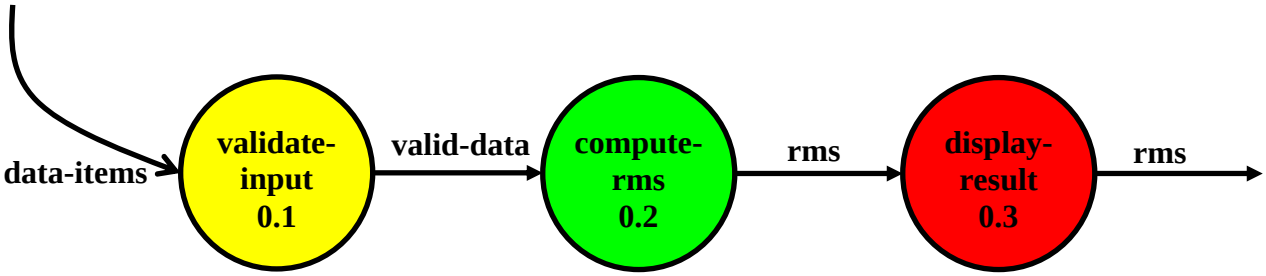


Fig. 36.11 Level 1 DFD

By observing the level 1 DFD, we identify the validate-input as the afferent branch, and write-output as the efferent branch, and the remaining (i.e. compute-rms) as the central transform. By applying the step 2 and step 3 of transform analysis, we get the structure chart shown in fig. 36.12.

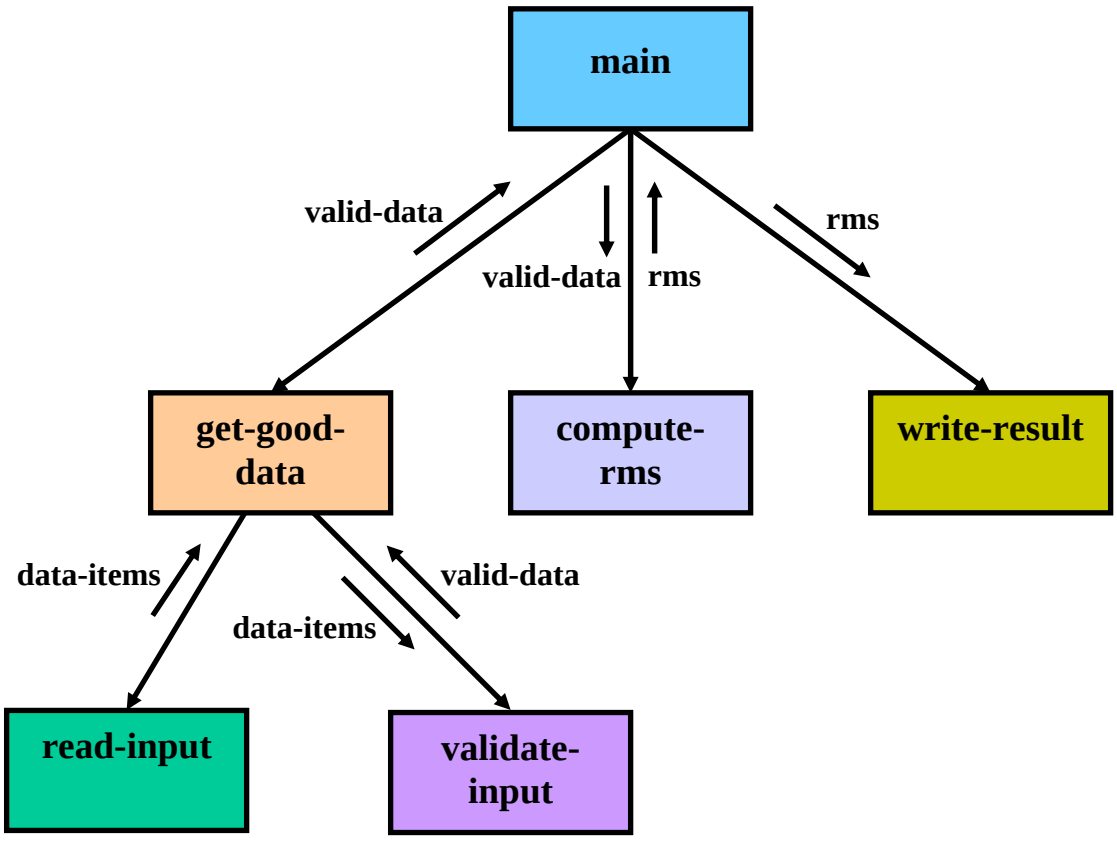


Fig. 36.12 Structure chart

2.4.4. Transaction Analysis

A transaction allows the user to perform some meaningful piece of work. Transaction analysis is useful while designing transaction processing programs. In a transaction-driven system, one of several possible paths through the DFD is traversed depending upon the input data item. This is in contrast to a transform centred system which is characterized by similar processing steps for each data item. Each different way in which input data is handled is a transaction. A simple way to identify a transaction is to check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transactions may not require any input data. These transactions can be identified from the experience of solving a large number of examples.

For each identified transaction, trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, draw a root module and below this module draw each identified transaction a module. Every transaction carries a tag, which identifies its type. Transaction analysis uses this tag to divide the system into transaction modules and a transaction-centre module.

The structure chart for the supermarket prize scheme software is shown in fig. 36.13.

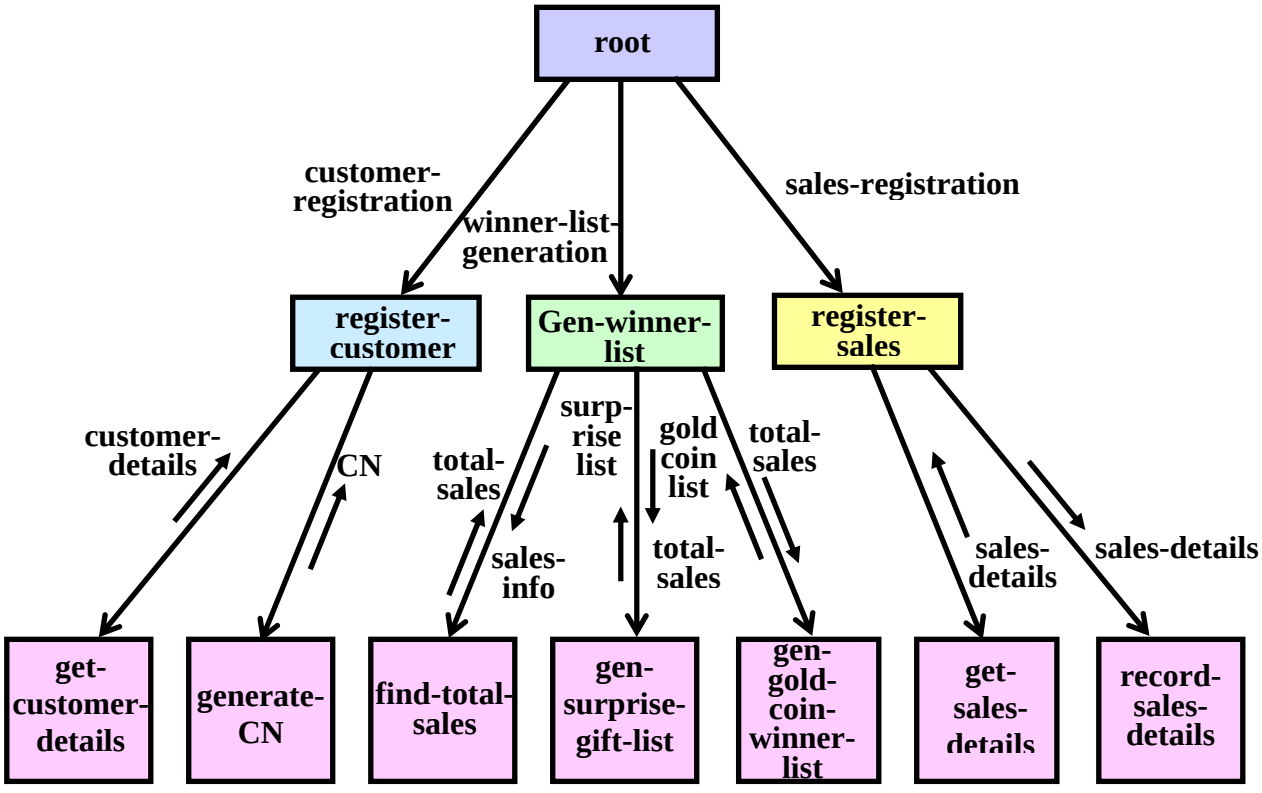


Fig. 36.13 Structure chart for the supermarket prize scheme

3. Exercises

1. Mark the following as True or False. Justify your answer.
   a. Coupling between two modules is nothing but a measure of the degree of dependence between them.
   b. The primary characteristic of a good design is low cohesion and high coupling.
   c. A module having high cohesion and low coupling is said to be functionally independent of other modules.
   d. The degree of coupling between two modules does not depend on their interface complexity.
   e. In the function-oriented design approach, the system state is decentralized and not shared among different functions.
   f. The essence of any good function-oriented design technique is to map the functions performing similar activities into a module.
   g. In the object-oriented design, the basic abstraction is real-world functions.
   h. An OOD (Object-Oriented Design) can be implemented using object-oriented languages only.
   i. A DFD model of a system represents the functions performed by the system and the data flow taking place among these functions.
   j. A data dictionary lists all data items appearing in the DFD model of a system but does not capture the composition relationship among the data.
   k. The context diagram of a system represents it using more than one bubble.
   l. A DFD captures the order in which the processes (bubbles) operate.
   m. There should be at the most one control relationship between any two modules in a properly designed structure chart.
2. For the following, mark all options which are true.
   a. The desirable characteristics that every good software design need are
      - Correctness
      - Understandability
      - Efficiency
      - Maintainability
      - All of the above
   b. A module is said to have logical cohesion, if
      - it performs a set of tasks that relate to each other very loosely.
      - all the functions of the module are executed within the same time span.
      - all elements of the module perform similar operations, e.g. error handling, data input, data output, etc.
      - None of the above.
   c. High coupling among modules makes it
      - difficult to understand and maintain the product
      - difficult to implement and debug
      - expensive to develop the product as the modules having high coupling cannot be developed independently
      - all of the above
   d. The desirable characteristics that every good software design need are
      - error isolation
      - scope of reuse

- understandability
- all of the above

e. The purpose of structured analysis is
   - to capture the detailed structure of the system as perceived by the user
   - to define the structure of the solution that is suitable for implementation in some programming language
   - all of the above

f. Structured analysis technique is based on
   - top-down decomposition approach
   - bottom-up approach
   - divide and conquer principle
   - none of the above

g. Data Flow Diagram (DFD) is also known as a:
   - structure chart
   - bubble chart
   - Gantt chart
   - PERT chart

h. The context diagram of a DFD is also known as
   - level 0 DFD
   - level 1 DFD
   - level 2 DFD
   - none of the above

i. Decomposition of a bubble is also known as
   - classification
   - factoring
   - exploding
   - aggregation

j. Decomposition of a bubble should be carried on
   - till the atomic program instructions are reached
   - up to two levels
   - until a level is reached at which the function of the bubble can be described using a simple algorithm
   - none of the above

k. The bubbles in a level-1 DFD represent
   - exactly one high-level functional requirement described in SRS document
   - more than one high-level functional requirement
   - part of a high-level functional requirement
   - any of the above depending on the problem

l. By looking at the structure chart, we can
   - say whether a module calls another module just once or many times
   - not say whether a module calls another module just once or many times
   - tell the order in which the different modules are invoked
   - not tell the order in which the different modules are invoked

m. In which of the following ways does a structure chart differ from a flow chart?
   - it is always difficult to identify the different modules of the software from its flow chart representation

- data interchange among different modules is not presented in a flow chart
- sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart
- none of the above

n. The input portion in the DFD that transforms input data from physical to logical form is called
- central transform
- efferent branch
- afferent branch
- none of the above

o. If during structured design, you observe that the data entering a DFD are incident on different bubbles, then you would use:
- transform analysis
- transaction analysis
- combination of transform and transaction analysis
- neither transform nor transaction analysis

p. During detailed design, which of the following activities take place?
- the pseudo code for the different modules of the structure chart are developed in the form of MSPECs
- data structures are designed for the different modules of the structure chart
- module structure is designed
- none of the above

3. State the major design activities. Identify separately, the activities undertaken during high-level design and detailed design.
4. Why is functional independence of a module a key factor for a good software design?
5. What the salient features of a function-oriented design approach and object-oriented design approach. Differentiate between both these approaches.
6. Identify the aim of the structured analysis activity. Which documents are produced at the end of structured analysis activity?
7. Identify the necessity of constructing DFDs in the context of a good software design.
8. Write down the importance of data dictionary in the context of good software design.
9. Explain the term “balancing a DFD” with an example
10. Discuss the essential activities required to develop the DFD of a system more systematically.
11. What do you understand by top-down decomposition in the context of structured analysis? Explain with a suitable example.
12. Identify the common errors made during construction of a DFD model. Identify the shortcomings of the DFD model.
13. Differentiate between a structure chart and a flow chart.
14. Explain transform analysis with a suitable example.
15. Explain transaction analysis with an example.