15 Function-oriented Design

Objectives

- To explain how a software design may be represented as a set of functions which share system state information.
- To introduce notations which may be used to represent a function-oriented design.
- To develop an example which illustrates the process of function-oriented design.
- To compare, using a common example, sequential and concurrent function-oriented design and object-oriented design.

Contents

A function-oriented design strategy relies on decomposing the system into a set of interacting functions with a centralised system state shared by these functions (Figure 15.1). Functions may also maintain local state information but only for the duration of their execution.

Function-oriented design has been practised informally since programming began. Programs were decomposed into subroutines which were functional in nature. In the late 1960s and early 1970s several books were published which described 'top-down' functional design. They specifically proposed this as a 'structured' design strategy (Myers, 1975; Wirth, 1976; Constantine and Yourdon, 1979). These led to the development of many design methods based on functional decomposition.

Function-oriented design conceals the details of an algorithm in a function but system state information is not hidden. This can cause problems because a function can change the state in a way which other functions do not expect. Changes to a function and the way in which it uses the system state may cause unanticipated changes in the behaviour of other functions.

A functional approach to design is therefore most likely to be successful when the amount of system state information is minimised and information sharing is explicit. Systems whose responses depend on a single stimulus or input and which are not affected by input histories are naturally functionally-oriented. Many transaction-processing systems and business data-processing systems fall into this class. In essence, they are concerned with record processing where the processing of one record is not dependent on any previous processing.

An example of such a transaction processing system is the software which controls automatic teller machines (ATMs) which are now installed outside many banks. The service provided to a user is independent of previous services provided so can be thought of as a single transaction. Figure 15.2 illustrates a simplified functional design of such a system. Notice that this design follows the centralised management control model introduced in Chapter 13.

In this design, the system is implemented as a continuous loop and actions are triggered when a card is input. Functions such as Dispense_cash, Get_account_number, Order_statement, Order_checkbook, etc. can be identified which implement system actions. The system state maintained by the program is minimal. The user services operate independently and do not interact with each other. An object-oriented design would be similar to this and would probably not be significantly more maintainable.

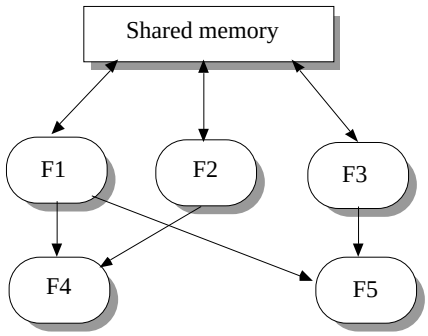


Figure 15.1 A function-oriented view of design

```
loop
    loop
    Print_input_message (" Welcome - Please enter your card") ;
    exit when Card_input ;
    end loop ;
    Account_number := Read_card ;
    Get_account_details (PIN, Account_balance, Cash_available) ;
    if Validate_card (PIN) then
    loop
     Print_operation_select_message ;
    case Get_button is
              when Cash_only =>
                   Dispense_cash (Cash_available, Amount_dispensed) ;
              when Print_balance =>
                   Print_customer_balance (Account_balance) ;
              when Statement =>
                   Order_statement (Account_number) ;
              when Check_book =>
                   Order_checkbook (Account_number) ;
    end case ;
    Eject_card ;
    Print ("Please take your card or press CONTINUE") ;
    exit when Card_removed ;
    end loop ;
    Update_account_information (Account_number,  Amount_dispensed) ;
     else
         Retain_card ;
    end if ;
end loop ;
```

Figure 15.2 The functional design of software for an ATM

As object-oriented design has become more widely used, some people have suggested that function-oriented design is obsolete. It should be superseded by an object-oriented approach. There are several good reasons why this should not and will not happen:

1. As discussed above, there are classes of system, particularly business systems, where object-oriented design does not offer significant advantages in terms of system maintainability or reliability. In these cases, an object-oriented approach may result in a less efficient system implementation.

2. Many organisations have developed standards and methods based on functional decomposition. They are understandably reluctant to discard these in favour of object-oriented design. Many design methods and associated CASE tools are functionally oriented. A large capital and training investment in these systems has been made.

3. An enormous number of systems have been developed using a functional approach. These legacy systems will have to be maintained for the foreseeable future. Unless they are radically re-engineered, their design structure will remain functional.

It is not sensible to view the function-oriented and object-oriented approaches as competing design strategies. Rather, they are applicable in different

circumstances and for different types of application. Good designers chose the most appropriate strategy for the application that is being developed rather than use a single approach.

In this chapter, I illustrate a function-oriented design process using a number of examples. The activities in that process are:

1. *Data-flow design* Model the system design using data-flow diagrams. This should show how data passes through the system and is transformed by each system function. This model may be derived from data-flow models developed during the analysis process.
2. *Structural decomposition* Model how functions are decomposed into sub-functions using graphical structure charts.
3. *Detailed design description* Describe the entities in the design and their interfaces. These descriptions may be recorded in a data dictionary. Also describe the control structure of the design using a program description language (PDL) which includes conditional statements and looping constructs.

As with all design processes, these activities are not carried out in sequence but are interleaved during the design process.

15.1 Data-flow design

Data-flow design is concerned with designing a sequence of functional transformations that convert system inputs into the required outputs. The design is represented as data-flow diagrams. These diagrams illustrate how data flows through a system and how the output is derived from the input through a sequence of functional transformations.

Data-flow diagrams are a useful and intuitive way of describing a system. They are normally understandable without special training, especially if control information is excluded. They show end-to-end processing. That is, the flow of processing from when data enters the system to where it leaves the system can be traced.

Data-flow design is an integral part of a number of design methods and most CASE tools support data-flow diagram creation. Different methods may use different icons to represent data-flow diagram entities but their meanings are similar. The notation which I use is based on the following symbols:

1. *Rounded rectangles* represent functions which transform inputs to outputs. The transformation name indicates its function.
2. *Rectangles* represent data stores. Again, they should be given a descriptive name.
3. *Circles* represent user interactions with the system that provide input or receive output.

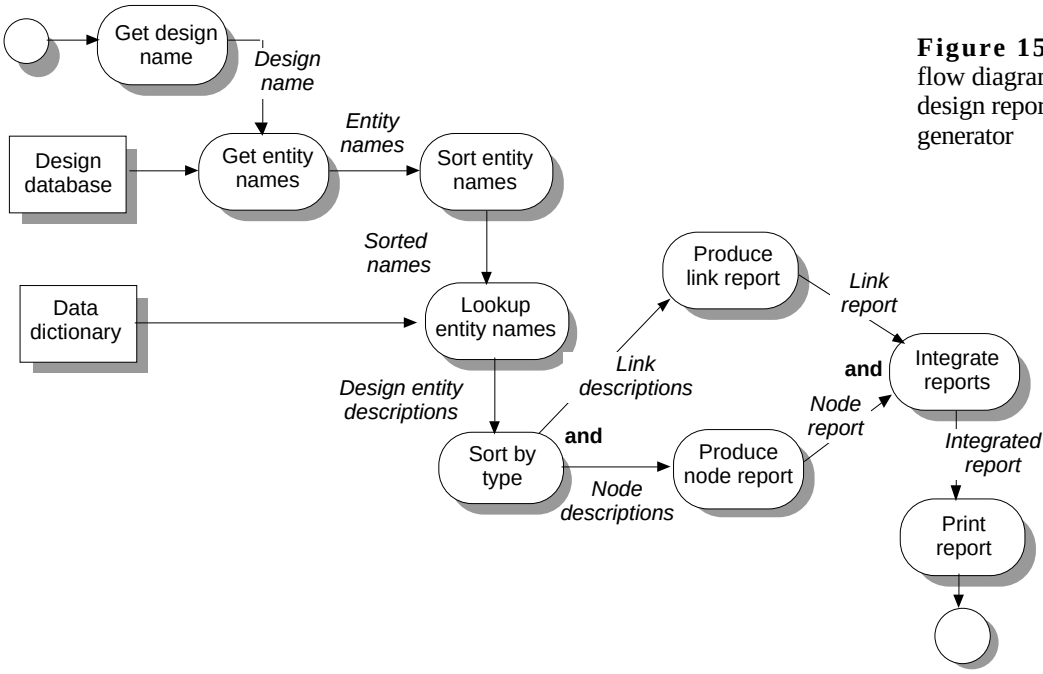


Figure 15.3 Data-flow diagram of a design report generator

4. *Arrows* show the direction of data flow. Their name describes the data flowing along that path.

5. T*he keywords* 'and' *and* 'or'. These have their usual meanings as in boolean expressions. They are used to link data flows when more than one data flow may be input or output from a transformation.

This notation is illustrated in Figure 15.3 which shows a data-flow design of a design report generator. The report generator produces a report which describes all of the named entities in a data-flow diagram. The user inputs the name of the design represented by the diagram. The report generator then finds all the names used in the data-flow diagram. It looks up a data dictionary and retrieves information about each name. This is then collated into a report which is output by the system.

Data-flow diagrams show functional transformations but do not suggest how these might be implemented. A system described in this way might be implemented as a single program using functions or procedures to implement each transformation. Alternatively, it could be implemented as a number of communicating tasks.

15.2 Structural decomposition

As well as a data-flow model of a system, it is also useful to develop a structural system model. This structural model shows how a function is realised by a number

of other functions which it calls. Structure charts are a graphical way to represent this decomposition hierarchy. Like data-flow diagrams, they are dynamic rather than static system models. They show how one function calls others. They do not show the static block structure of a function or procedure.

A function is represented on a structure chart as a rectangle. The hierarchy is displayed by linking rectangles with lines. Inputs and outputs (which may be implemented either as parameters or shared variables) are indicated with annotated arrows. An arrow entering a box implies input, leaving a box implies output. Data stores are shown as rounded rectangles and user inputs as circles.

Converting a data-flow diagram to a structure chart is not a mechanical process. It requires designer insight and creativity. However, there are several 'rules of thumb' which may be applied to help designers assess if their decomposition is likely to be a reasonable one:

1. Many systems, particularly business systems for which functional design is most appropriate, can be considered as three-stage systems. These are input some data, perhaps with validation and checking, process the data then output the data, perhaps in the form of a report or perhaps to some other file. A master file may also be updated. The first-level structure chart may therefore have 3 or 4 functions corresponding to input, process, master-file update and output. Figure 15.4 illustrates this structure although, in this case, there is no master file processing.
2. If data validation is required, functions to implement these should be subordinate to an input function. Output formatting, printing and writing to disk or tape should be subordinate to an output function.
3. The role of functions near the top of the structural hierarchy may be to control and coordinate a set of lower-level functions.
4. The objective of the design process is to have loosely coupled, highly cohesive components (cohesion and coupling were discussed in Chapter 10). Functions should therefore do one thing and one thing only.
5. Each node in the structure chart should have between two and seven subordinates. If there is only a single subordinate, this implies that the unit represented by that node may have a low degree of cohesion. The component may not be single function. A single subordinate means that another function has been factored out. If a node has too many subordinates, this may mean that the design has been developed to too low a level at that stage.

Three process steps, which follow these guidelines, can be identified for the transformation process from data-flow diagram to structure chart:

1. *Identify system processing transformations* These are the transformations in the diagram which are responsible for central processing functions. They are not concerned with any input or output functions such as reading or writing data, data validation or filtering or output formatting. Group these transformations under a single function at the first-level in the structure chart.

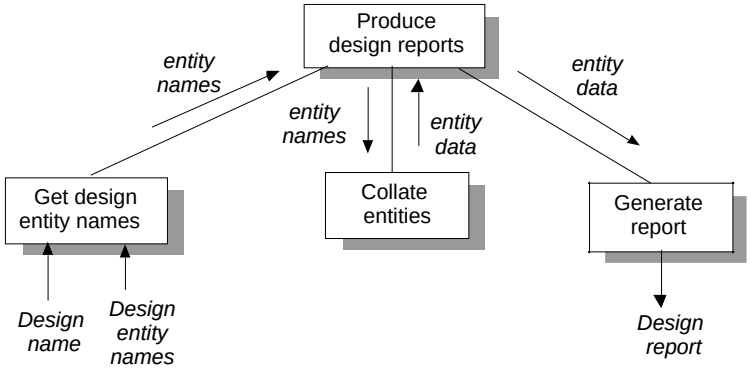


Figure 15.4 Initial structure chart for the design report generator

2. *Identify input transformations* These are concerned with reading data, checking it, removing duplicates, etc. These should also be grouped under a single function at the first-level in the structure chart.

3. *Identify output transformations* These are transformations which prepare and format output or write it to the user's screen or other device. However, there are several 'rules of thumb' that can be applied to help designers in this process.

In the design report generator data-flow diagram (Figure 15.3), the processing functions are those which sort the input, look up the data dictionary and sort the information retrieved from the data dictionary. In the structure chart (Figure 15.4) these are collected together into a single function called Collate entities. This initial structure chart can be decomposed to show subordinate functions which also reflect transformations in the data-flow diagram (Figure 15.5).

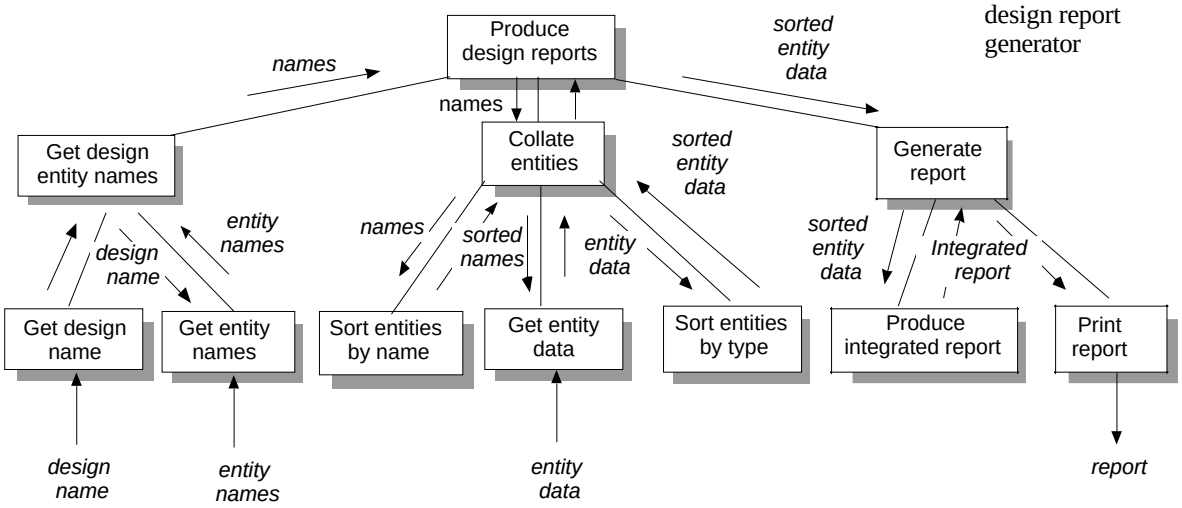


Figure 15.5 Second-level structure chart for the design report generator

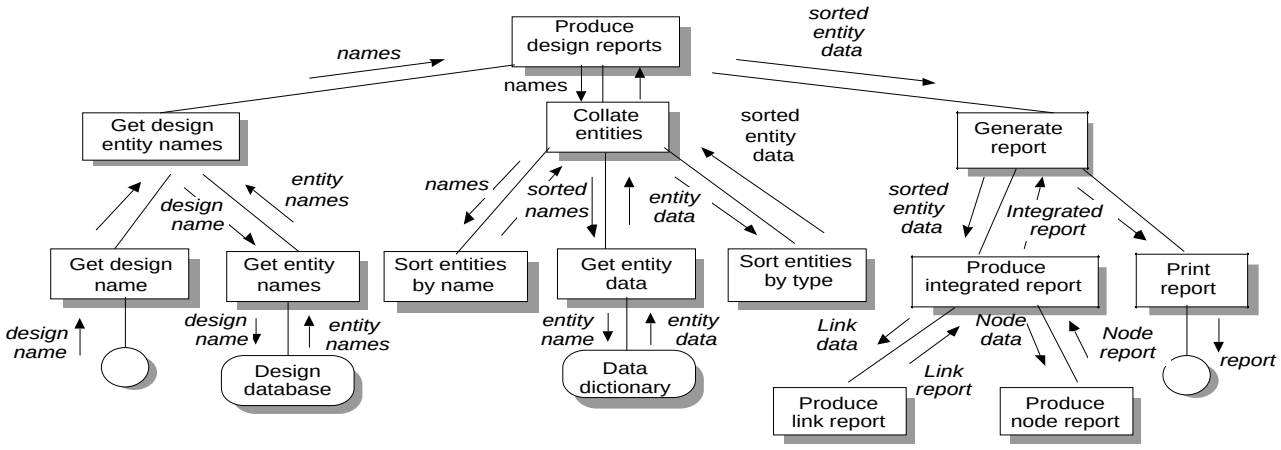


Figure 15.6 Final structure chart for the design report generator

The input transformations stop at the function Sort entity names. In this case they are simply concerned with reading information. No data checking is required. They are represented as a function called Get design entity names.

The output transformations are concerned with producing reports for link and node type, report integration and printing. These are all concerned with the formatting and organisation of design entity descriptions. These are grouped under the function Generate report.

All the principal routines in the design report generator have now been identified. It only remains to add a final level of more detailed routines and to show how the system data stores are accessed. The final structure chart is shown in Figure 15.6. This chart does not show the access functions to the data dictionary and the design database. I assume that these data stores are like abstract data types and have their own access functions.

Other components may be included in a structure chart which are not directly concerned with data transformation. Because they do not transform data, they do not appear on the data-flow diagram. For example, components which are concerned with logging-in and logging-out a user, system initialisation and any other components concerned with system control rather than data processing may be included at the structural decomposition level.

15.3 Detailed design description

At this stage in the design process, the designer should know the organisation of the design and what each function should do. Design entity description is concerned

with producing a short design specification (sometimes called a *minispec*) of each function. This requires describing the function, its inputs and its outputs.

Making this information explicit usually reveals flaws in the initial decomposition or functions which have been omitted. The data-flow diagrams and structure charts must be re-visited and modified to incorporate the improved understanding of the design.

The best way to manage these functional descriptions is to maintain them in a data dictionary. Data dictionaries were introduced in Chapter 6 as a way of recording information in system models developed as part of the requirements analysis process. They can also be used to record information about design entities. Maintaining names and descriptions in a data dictionary reduces the chances of mistakenly reusing names and provides design readers with insights into the designer's thinking.

Data dictionary entries can vary in detail from a short informal description to a specification of the function in a design description language. Figure 15.7 shows some of the data dictionary entries that might be made for the design report generator. As you can see, it includes information about data as well as the functions in the system.

Some CASE tools may include facilities which allow data dictionaries to be accessed at the same time as a design diagram. This allows information about individual entities to be viewed at the same time as the diagram showing all entities and their relationships. The tool may allow a design entity to be selected then display the corresponding information from the data dictionary. Figure 15.8 is an example of this facility. It shows a pop-up window which includes a form describing the selected transform in the data-flow diagram.

Entity name	Type	Description
Design name	STRING	The name of the design assigned by the design engineer.
Get design name	FUNCTION	*Input*: Design name *Function*: This function communicates with the user to get the name of a design that has been entered in the design database. *Output:* Design name
Get entity names	FUNCTION	*Input*: Design name *Function*: Given a design name, this function accesses the design database to find the names of the entities (nodes and links) in that design. *Output*: Entity names
Sorted names	ARRAY of STRING	A list of the names of the entities in a design held in ascending alphabetical order.

Figure 15.7 Data dictionary entries for the design report generator

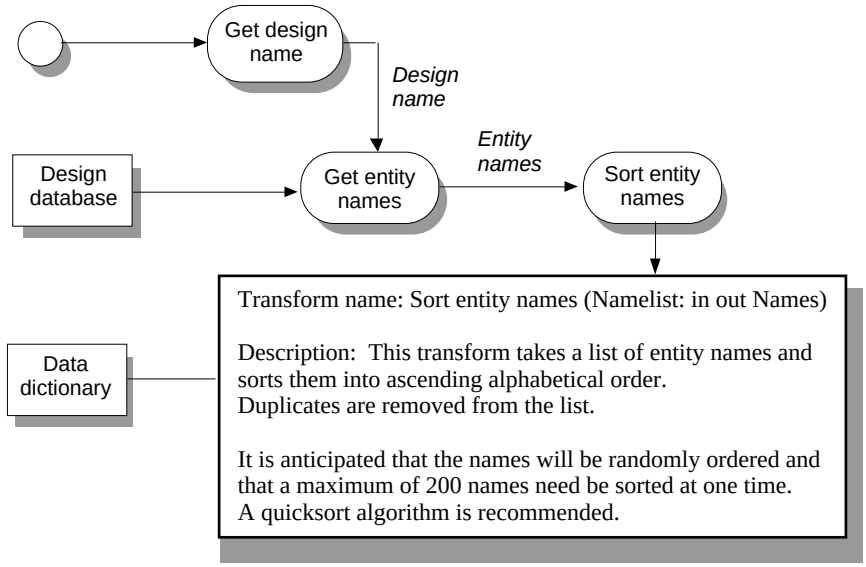
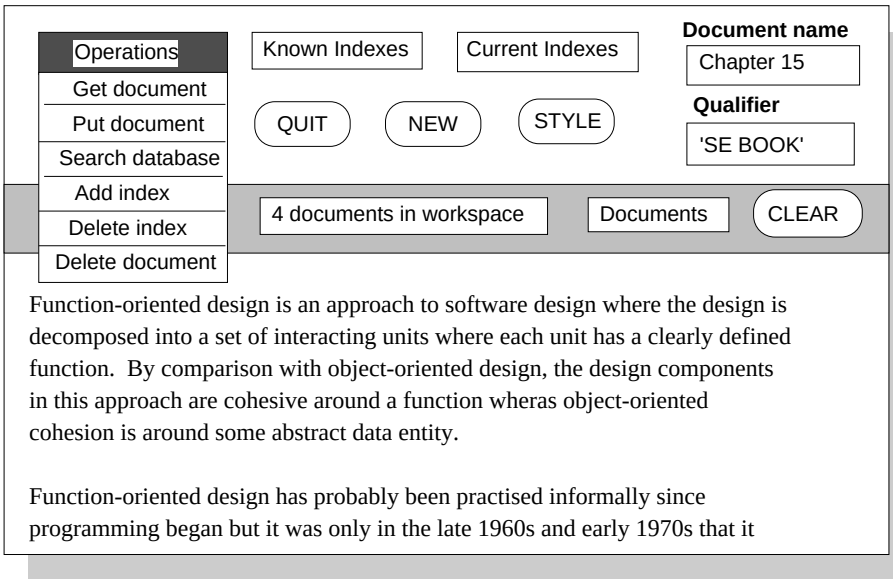


Figure 15.8 Information about design entities from a data dictionary

The next stage of the functional design process is to produce detailed designs for each part of the design. These detailed designs should include control information and more precise information about the data structures manipulated.

Figure 15.9 The OIRS user interface



These detailed designs may be expressed using some program description language, in some more detailed graphical notation or directly in a programming language. I have already shown examples of how a PDL may be used to describe a design in Figure 15.2. Further examples are shown in Figures 15.11, 15.14 and 15.16.

15.4 A comparison of design strategies

In Chapter 14, I introduced an example of an office information system. This was used to illustrate object identification using a grammatical analysis of a system description. In this section, this example will be expanded and used to compare functional and object-oriented approaches to design. I will also show how a design can be developed for the same system using concurrent processes.

The Office Information Retrieval System (OIRS) is an office system which can file documents in one or more indexes, retrieve documents, display and maintain document indexes, archive documents and delete documents. Users request operations from a menu-based interface and the system always returns a message to the user indicating the success or failure of the request.

The interface to this system is a form which has a number of fields (Figure 15.9). Some of these fields are menu fields where the user can choose a particular option. Other fields allow user textual input. Menu items may be selected by pointing with a mouse or by moving a cursor using keyboard commands.

The fields displayed in the form are as follows:

1. *The operation field* Selecting this field causes a menu of allowed operations to be displayed as shown in Figure 15.9.
2. *The known indexes field* Selecting this field causes a menu of existing index names to be displayed. Selecting an item from this list adds it to the current index list.
3. *The current indexes field* Selecting this field displays a list of the indexes where the current document is to be indexed.
4. *The document name field* This specifies the name of the document to be filed or the name of the document to be retrieved. If this name is not filled in, the user is prompted for a value.
5. *The qualifier field* This is a pattern which is used in searching. For example, the pattern 'A-K' may be specified with a command to look up the names of documents in the current index lists. The qualifier causes only those names which begin with a letter from A to K to be listed. Alternatively, the qualifier field might contain a keyword such as 'Software Engineering'. An index search retrieves all documents which contain this keyword.
6. *The current workspace* Documents are retrieved to the current workspace that may contain several documents. The user may choose a document in the workspace by selecting its name from the workspace menu. Clicking on the Clear button in the workspace control bar removes the selection from the

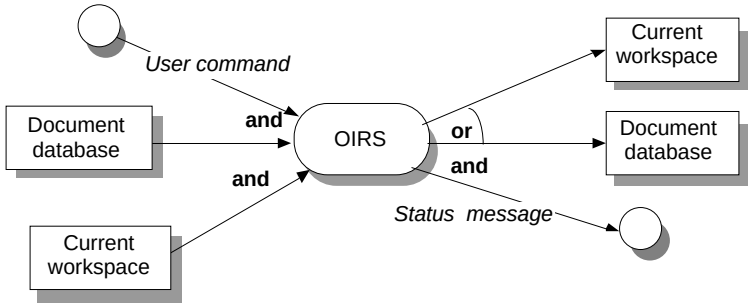


Figure 15.10 Inputs and outputs of the OIRS

workspace. Moving the cursor into the workspace causes the system to enter document edit mode.

When developing a functional design, the initial stage should treat the system as a black box. It should simply show the inputs and outputs of the system with the system itself represented as a single transformation (Figure 15.10). The arc annotated with 'or' joining the output data flows means that one or the other but not both of these data flows occur. Notice that the document database and the current workspace are both inputs and outputs.

Functional designs of interactive systems often follow a common detailed architectural model . This is a command 'fetch and execute' loop. Figure 15.11 describes the general form of a fetch-execute loop. The ATM design, described in Figure 15.2, is an example of this model.

In this design, the Get command function has been separated from the Execute command function to allow for different types of user interface and user interface evolution. If the system is moved from a computer without a mouse to a computer with a mouse (say) only the Get command function need be changed. It is generally good design practice to separate, as far as possible, the user interface of a system from the data processing functions. This is discussed in Chapter 17.

A data-flow diagram for the OIRS based on this model is shown in Figure 15.12. The central transform is concerned with command execution and further functions are incorporated to manage the database and the workspace.

```
procedure Interactive_system is
begin
   loop
        Command := Get_command;
        if Command = "quit" then
             -- Make sure files etc. are closed properly
             Close_down_system ;
             exit ;
        else
             Input_data := Get_input_data ;
             Execute_command (Command, Input_data, Output_data) ;
        end if ;
   end loop ;
end Interactive_system ;
```

Figure 15.11 Fetch-execute model of an interactive system

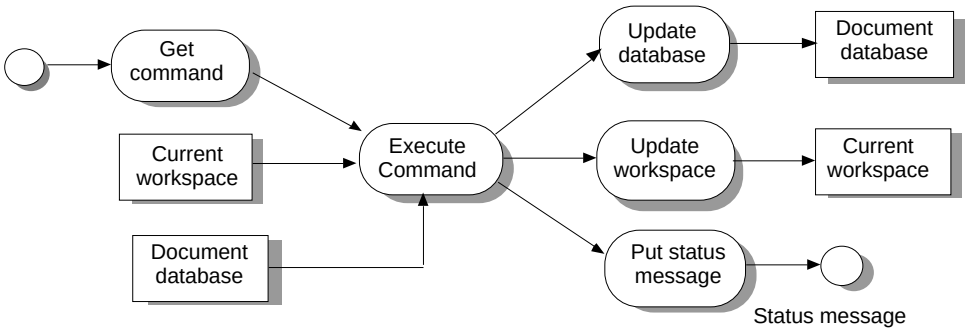


Figure 15.12 Top-level data-flow design for the OIRS

The Execute Command transformation can now be considered in isolation and can be decomposed into simpler transformations (Figure 15.13).

Figure 15.13 shows that there are three different classes of command in the OIRS. These are commands to update the workspace (Get document, Search database, Clear workspace and the implicit Edit command), commands to update the database (Put document, Delete document) and index commands (Add index, Delete index, Display index lists).

For brevity, I have skipped the structural decomposition and entity description stages in the functional design process and have gone straight to detailed design description. Figure 15.14 is a design, expressed in a PDL, for the top-level of the OIRS. As discussed in the previous section, additional functions have been included for login and initialisation which are not shown in the data-flow diagram.

Note that the control structure design shown in Figure 15.14 is not a syntactically correct program. It does not include declarations of the entities which are used. As discussed above, these names should be entered in a data dictionary along with a description of the entities which they identify.

```
procedure OIRS is
begin
    User := Login_user ;
     Workspace := Create_user_workspace (User) ;
    -- Get the users own document database using the user id
    DB_id := Open_document_database (User) ;
    -- get the user's personal index list;
    Known_indexes := Get_document_indexes (User) ;
    Current_indexes := NULL ;
    -- command fetch and execute loop
     loop
         Command := Get_command ;
         exit when Command = Quit ;
         Execute_command ( DB_id, Workspace, Command, Status) ;
         if Status = Successful then
              Write_success_message ;
         else
              Write_error_message (Command, Status) ;
         end if ;
    end loop ;
    Close_database (DB_id) ;
    Logout (User) ;
end OIRS ;
```

Figure 15.14 High-level design description of the OIRS

The design process continues by decomposing each of the design components in more detail. This decomposition should stop when the design has been described at a sufficiently detailed level that a program can be developed in the chosen implementation language. This level of detail will obviously vary depending on the language used. The lower level the implementation language, the more detail is

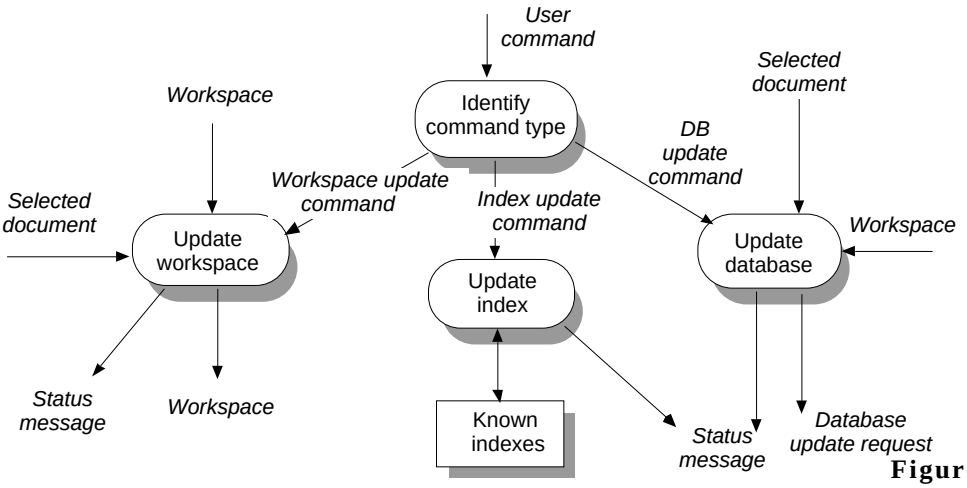


Figure 15.13 Data-flow diagram for Execute command

required in the design.

15.4.1 Concurrent systems design

The above functional design models the OIRS as a sequential system with a single control loop which fetches and executes commands in sequence. An alternative approach to the detailed design is to implement the system as a number of concurrent process. As data-flow diagrams explicitly exclude control information, they can also be the starting point for a concurrent design. A standard implementation technique for real-time systems (see Chapter 16) is to take a data-flow diagram and to implement each of its transformations as a separate process.

In this case, implementing each data-flow transformation as a separate process would not be an efficient way of designing the system. There is a scheduling overhead involved in starting and stopping processes and managing process communications. Unless it is absolutely necessary because of real-time requirements, it is best to avoid decomposing a system into many small processes.

However, to improve real-time response in window-based systems, it is often necessary to identify those parts of the system which must respond to user events and implement these as separate processes. This allows the system to be responsive to time-critical user events such as mouse movements. The strict sequence which is forced by a fetch-execute cycle can be avoided. In the OIRS, there are two situations where user actions cause an event to be generated:

1. When a command is selected. The user moves the cursor into a menu and makes a choice. An event is generated informing the system that a command has been chosen.

2. When the cursor is moved into the workspace and the user starts to type.

The user can move the cursor between these areas at any time so the system must be able to cope with the change. This can be handled in a fetch-execute system by making sure that the workspace editor keeps track of the cursor but some code duplication in such a situation is inevitable.

Given that the input processing (commands and text) is to be implemented using separate processes, it then makes sense to identify corresponding processes to handle command execution and output processing. This leads to a set of four processes as shown in Figure 15.15. Control passes from process to process in response to events such as a cursor being positioned in a workspace, a command menu being selected, etc.

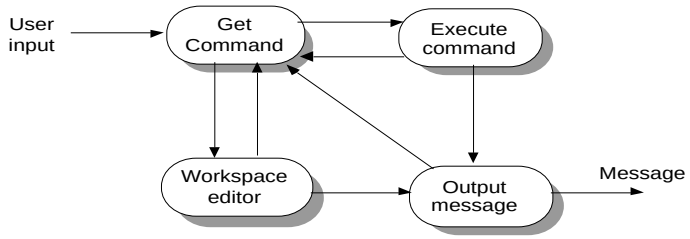


Figure 15.15 Process decomposition of the OIRS

The Get command task tracks the mouse and responds to low-level mouse events. When a command area is selected, it initiates the command execution process. Similarly, the command execution process produces status messages which are processed by the output task. Workspace editing is also implemented as a parallel task. The editor is initiated and suspended as the cursor is moved in and out of the workspace window.

The detailed design of the overall system and the Get command process is shown, in Ada, in Figure 15.16. This uses Ada's facilities (tasks) for implementing parallel systems. Ada task entries correspond to procedure calls. Therefore, calling an entry Display indexes in Execute command is like calling a function of that name. Details of Ada tasking can be found in Ada textbooks (Burns and Wellings, 1990; Barnes, 1994).

```
procedure Office_system is
   task Get_command ;
   task Process_command is
        entry Command_menu ;
        entry Display_indexes ;
        entry Edit_qualifier ;
        -- Additional entries here. One for each command
    end Process_commands ;
   task Output_message is
        entry Message_available ;
   end Output_message ;
   task Workspace_editor is
   entry Enter ;
   entry Leave ;
   end Workspace_editor ;

   task body Get_command is
   begin
   -- Fetch/execute loop
   loop
   loop
              Cursor_position := Get_cursor_position ;
        exit when cursor positioned in workspace or
        (cursor positioned over menu and button pressed)
        Display_cursor_position ;
   end loop ;
   if In_workspace (Cursor_position) then
              Workspace_editor.Enter ;
   elsif In_command_menu (Cursor_position) then
              Process_command.Command_menu ;
   elsif In_Known_indexes (Cursor_position) then
              Process_command.Display_indexes ;
   elsif In_Current_indexes (Cursor_position) then
                        ...
                        Other commands here
                   ...
   end loop ; -- Fetch/execute
   end Get_command ;
-- other task implementations here
end Office_system ;
```

Figure 15.16 Detailed process design in Ada

15.4.2 Object-oriented design

Both the sequential and concurrent designs of the OIRS are functional because the principal decomposition strategy identifies functions such as Execute command, Update index, etc. By contrast, an object-oriented design focuses on the entities in the system with the functions part of these entities. There is not enough space here to develop a complete object-oriented design but I show the entity decomposition in this section. Notice that this is quite different from the functional system decomposition.

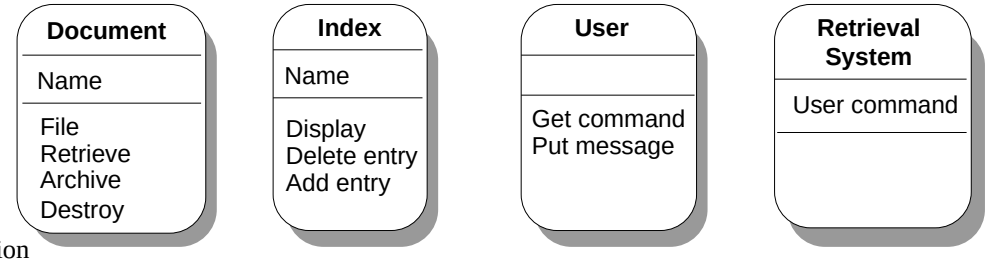


Figure 15.17 Initial decomposition of the OIRS into classes and objects

We have already seen an initial object identification for the OIRS in Chapter 14. The objects identified from that description are shown in Figure 15.17.

This object decomposition leaves out entities such as the database, the workspace, etc. The reason for this is that this decomposition was based on a simple description. This description did not describe all system facilities. Nor did it describe how these facilities are presented to users. We therefore need to define further system objects as shown in Figure 15.18.

Documents which are being used are held in a workspace and referenced in indexes. We therefore need objects which correspond to the workspace and each index. Index list is an object class used to create different lists of indexes. The document database is also represented as an object.

The initial object decomposition was based on a simple system description. With further information we now have to revisit this decomposition and see if it is still appropriate. We know that the user interface is a graphical interface with various fields and buttons so this should be represented as an object. It can serve as a replacement for the User object so this can now be discarded and its operations incorporated into Display. As a general design guideline, displays map onto objects. Object attributes represent fields on the display. We can see an example of this in Figure 15.19 which should be compared with Figure 15.9 which shows the OIRS display.

The Retrieval system object is also modified to add attributes representing the workspace and the indexes used. This object does not have any associated operations. The object is an active object which controls other objects but does not provide services to them. All its functionality is therefore concealed.

The principal objects in the OIRS have now been identified. The next stage of the design process is to set out object aggregation hierarchies and how these objects interact as discussed in Section 14.3. In essence, the fetch-execute loop as

Figure 15.18 Additional OIRS system objects

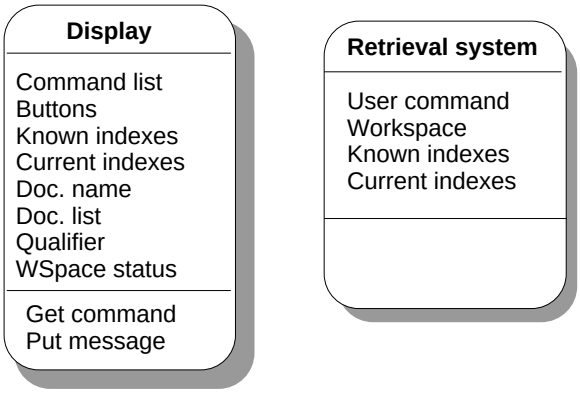


Figure 15.19 Modified OIRS objects

discussed above is still part of the design. Commands are fetched from the display object. Services provided by the workspace, document and index objects are called on to execute these commands.

Given the limited information here, it is impossible to say which design strategy is the best one for this system. The functional approach to design using data-flow diagrams is probably a more intuitive way to describe the overall design. However, the focus on function means that the reader of the design finds it more difficult to understand the entities which are manipulated by the system.

The object-oriented approach solves this difficulty as it is very good for entity description. Because of information encapsulation, object-oriented design generally leads to more robust systems where system data is less likely to be corrupted in the event of a program error. It may also be easier to change although this depends on the nature of the change. However, as Fichman and Kemerer (Fichman and Kemerer, 1992) point out, decomposing a system into loosely coupled objects often means that there is no overall picture of end-to-end processing in the design.

Designers must use the most appropriate approach to design. They should not be forced into the straitjacket of any particular design method or strategy. My own preference in this instance is to take a heterogeneous approach to the design. The basic functional fetch-execute loop should be retained and data-flow diagrams used to illustrate and understand the system's processing. The display management should be implemented as a separate process. The objects identified in this section should be used and referenced from the fetch-execute loop.

KEY POINTS

- Function-oriented design relies on identifying functions which transform their inputs to create outputs. In most systems, functions share some global system state.
- Many business systems which process transactions are naturally functional. Furthermore, there is a huge amount of legacy code which has been designed

using this approach. For these reasons, function-oriented design will continue alongside object-oriented design as an important design strategy.

- The functional design process involves identifying data transformations in the system, decomposing functions into a hierarchy of sub-functions, describing the operation and interface of each system entity and documenting the flow of control in the system.
- Data-flow diagrams are a means of documenting end-to-end data flow through a system. They do not include control information. Structure charts are a way of representing the hierarchical organisation of a system. Control may be documented using a program description language (PDL).
- Data-flow diagrams can be implemented directly as a set of cooperating sequential processes. Each transform in the data-flow diagram is implemented as a separate process. Alternatively, they can be realised as a number of procedures in a sequential program.
- Functional design and object-oriented design usually result in totally different system decompositions. However, the most appropriate design strategy is often a heterogeneous one where both functional and object-oriented approaches are used.

FURTHER READING

Software Design This book is a good general survey of software design techniques. Its orientation is towards functional approaches to design for the valid reason that these are more mature than object-oriented approaches. (D. Budgen, 1993, Addison-Wesley)

'Object-Oriented and Conventional Analysis and Design Methodologies' This is an excellent comparison of object-oriented and functional approaches to design. Its conclusion is that the claimed advantages of object-oriented design have not been conclusively demonstrated. (*IEEE Computer, 25 (10), October 1992)*

EXERCISES

15.1 Using examples, describe how data-flow diagrams may be used to document a system design. What are the advantages of using this type of design model?

15.2 Draw possible data-flow diagrams of system designs for the following applications. You may make any reasonable assumptions about these applications.

- Part of an electronic mail system which presents a mail form to a user, accepts the completed form and sends it to the identified destination.
- A salary system which computes employee salaries and deductions. The input is a list of employee numbers who are to be paid that month. The system maintains tables holding tax rates and the annual salary for each employee. The output is a salary slip for each employee plus a list of automated payments to be made by the company's bank.

15.3 Modify the design of the report generator shown in Figure 15.3 so that it becomes an interactive system. The user may give a design entity name and the report generator provides information about that entity. Alternatively, the user may provide a type name and the report generator produces a report about each entity of that type in a design. Document your modified design using data-flow diagrams and structure charts.

15.4 Convert the data-flow diagram of the report generator system described in Figure 15.3 into a design consisting of concurrent processes.

15.5 Using a design description language, describe a possible design for the design report generator whose data-flow diagram is given in Figure 15.3 and structure chart in Figure 15.7.

15.6 Explain how data dictionaries may be used to supplement design information in data-flow diagrams and structure charts.

15.7 Develop the data-flow diagrams shown in Figure 15.12 so that all transforms are documented with more detailed data-flow diagrams.

15.8 Develop the design of Execute Command in the office information retrieval system and describe its detailed design in a design description language.

15.9 Develop function-oriented designs for the systems described in Exercise 14.6. What are the principal differences between these and object-oriented designs?