15

15 Design and documentation : 1

The examples given in earlier chapters, particularly Chapter 12, have in a rather informal way illustrated a design style known as "top-down functional decomposition".

"Top down functional decomposition" is a limited approach to design. It works very well for simple scientific and engineering applications that have the basic structure:

```
initialize
get the input data
process the data
print the results
```

and where the "data" are something simple and homogeneous, like the array of double precision numbers in the heat diffusion example. Top down functional decomposition is not a good strategy for the overall design of more complex programs, such as those that are considered in Parts IV and V. However, it does reappear there too, in a minor role, when defining the individual "behaviours of objects".

Although the approach is limited, it is simple and it does apply to a very large number of simple programs. So, it is worth learning this design approach at least as a beginning.

The first section in this chapter simply summarizes materials from earlier examples using them to illustrate a basic strategy for program development. The second section looks briefly at some of the ways that you can document design decisions. Here, simple textual documentation is favoured.

15.1 TOP DOWN FUNCTIONAL DECOMPOSITION

As it says, you start the top with "program", decompose it into functions, then you iterate along the list of functions going down one level into each to decompose into auxiliary functions. The process is repeated until the auxiliary functions are so simple that they can be coded directly.

Note the focus on functions. You don't usually have to bother much about the data because the data will be simple – a shared array or something similar.

Beginning

You begin with a phrase or one sentence summary that defines the program:

- The program models a two-dimensional heat diffusion experiment. e.g..
- The program plays the game of "hangman".

and try to get a caricature sketch of the `main()` function. This should be some slight variation of the code at the start of this chapter.

Figure 15.1 illustrates initial decompositions for `main()` in the two example programs.

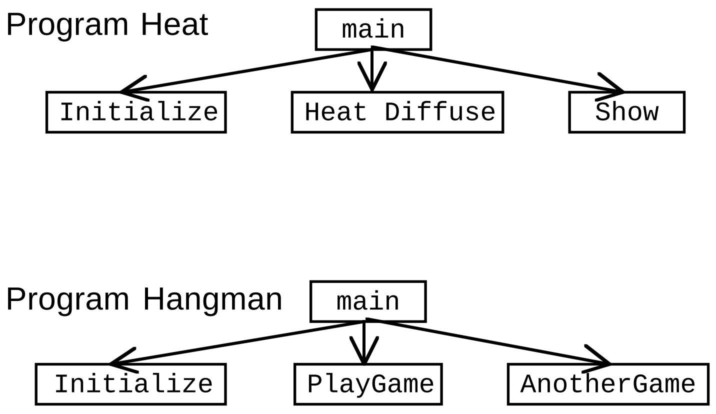

Figure 15.1 From program specification to initial functional decomposition.

The sketched functions are:

Program Heat's `main()`:

```
get iteration limit and print frequency
call Initialize() to initialize grid
loop
    call HeatDiffuse() to update values in grid
    if time to print, call Show()
```

Program Hangman's `main()`:

```
Initialize()
```

```
do
    PlayGame()
while AnotherGame()
```

Products of beginning step

You aim for this first stage is to get this initial sketch for `main()` and a list of "top level functions", each of which should be characterized by a phrase or sentence that summarizes what it does:

```
AnotherGame()
    Get and use a Yes/No input from user,
    return "true" if Yes was input

PlayGame()
    Organize the playing of one complete hangman game

HeatDiffuse()
    Recalculate temperature at each point on grid.
```

Second step

You now consider each of these top-level functions in isolation. It is all very well to say "Organize playing of game" or "Recalculate temperatures" but what do these specifications really mean.

It is essentially the same process as in the first step. You identify component operations involved in "organizing the playing of the game" or whatever, and you work out the sequence in which these operations occur. This gives you a caricature sketch for the code of the top-level function that you are considering and a list of second level functions. Thus, `HeatDiffuse()` gets expanded to:

```
HeatDiffuse
    get copy of grid values
    double loop
    for each row do
           for each col do
                  using copy work out average temp.
                         of environment of grid pt.
                  calculate new temp based on current
                         and environment
                  store in grid
    reset centre point to flame temperature
```

with a number of additional functions identified: `CopyGrid()`, `Averageof-Neighbors()`, and `NewTemperature()`.

The products of this design step are once again the sketches of functions and the lists of the additional auxiliary functions together with one sentence specifications for each:

```
CopyGrid()
    Copy contents of grid into separate data area.

AverageofNeighbors()
```

```
For a given grid point, this finds the average of the
temperatures of the eight neighboring points.
```

N-steps

This process is repeated until no more functions need be introduced. The relationships amongst the functions can be summarized in a "call graph" like that shown in Figure 15.2.

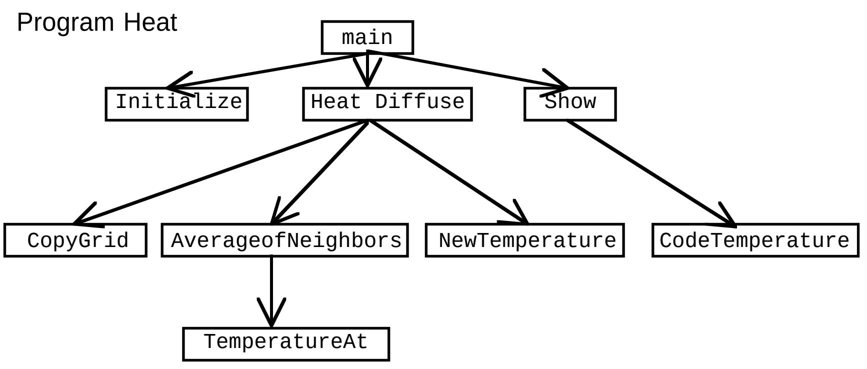

Figure 15.2 "Call graph" for a program.

Sorting out data and communications

The next stage in the design process really focuses on data. You have to resolve how the functions communicate and decide whether they share access to any global (or filescope) data. For example, you may have decided that function `TemperatureAt()` gets the temperature at a specified grid point, but you have still to determine how the function "knows" which point and how it accesses the grid.

The "input/output" argument lists of all the functions, together with their return types should now be defined. A table summarizing any filescope or global data should be made up.

Planning test data

Generally, you need to plan some tests. A program like Hangman does not require any elaborate preplanned tests; the simple processing performed can be easily checked through interaction with the program. The Heat program was checked by working out in advance the expected results for some simple cases (e.g. the heated point at 1000°C and eight neighbors at 25°C) and using the debugger to check that the calculated "new temperatures" were correct. Such methods suffice for simple programs but as you get to move elaborate programs, you need more elaborate preplanned tests. These should be thought about at this stage, and a suitable test plan composed.

Finalising the design

The final outputs that you want from the design process will include:

1. A sketch for `main()` summarising the overall processing sequence.

2. A list of function prototypes along with one sentence descriptions of what these functions do.

```
char CodeTemperature(double temp);
       Converts temperature to a letter code; different
       letters correspond to different temperature ranges.

void Show(const Grid g);
       Iterates over grid printing contents row by row,
       uses CodeTemperature() to convert temp. to letter.
```

3. A list of any typedef types (e.g. `typedef double Grid[kROWS][kCOLS]`) and a list defining constants used by the program.

4. A table summarizing global and filescope data.

5. A more detailed listing of the functions giving the pseudo-code outlines for each.

6. A test plan.

It is only once you have this information that it becomes worth starting coding.

Although you will often complete the design for the entire program and then start on implementation, if the program is "large" like the Hangman example then it may be worth designing and implementing a simplified version that is then expanded to produce the final product.

15.2 DOCUMENTING A DESIGN

For most programs, the best documentation will be the design outlines just described. You essentially have two documents. The first contains items 1…4. It is a kind of executive summary and is what you would provide to a manager, to coworkers, or to the teaching assistant who will be marking your program. The second document contains the pseudo-code outlines for the functions. This you will use to guide your implementation. Pseudo-code is supposed to be easier to read than actual code, so you keep this document for use by future "maintenance programmers" who may have to implement extensions to your code (they should also be left details of how to retest the code to check that everything works after alterations).

This documentation is all textual. Sometimes, diagrams are required. Call graphs, like that in Figure 15.2, are often requested. While roughly sketched call graphs are useful while you are performing the design process and need to keep records as you move from stage to stage, they don't really help that much in documenting large programs. Figure 15.3 illustrates some of the problems with "call graphs".

The call graph for the sort program using Quicksort may be correct but it really fails to capture much of the behaviour of that recursive system. The call graph for the Hangman program is incomplete. But even the part shown is overwhelming in its complexity.

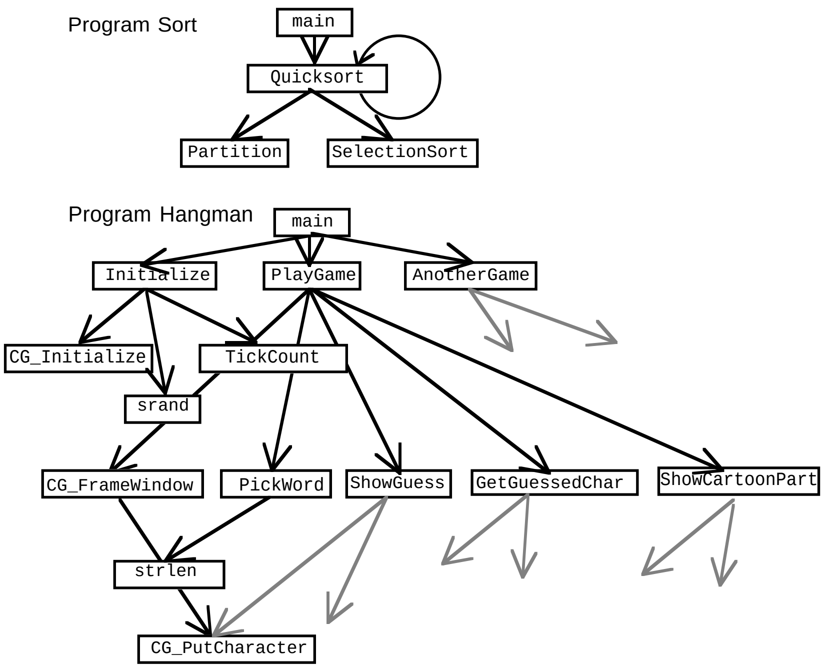

Figure 15.3 Further examples of "call graphs".

Psychologists estimate that the human brain can work with about seven items of information; if you attempt to handle more, you forget or confuse things. A complete call graph has too much information present and so does not provide a useful abstract overview.

You could break the graph down into subparts, separating PlayGame's call graph from the overall program call graph. But such graphs still fail to convey any idea of the looping structure (the fact that some functions are called once, others many times).

Overall, call graph diagrams aren't a particularly useful form of documentation.

Sometimes, "flowcharts" are requested rather than pseudo-code outlines for functions. "Flowcharts" are old; they are contemporary with early versions of FORTRAN. Flowcharting symbols correspond pretty much to the basic programming constructs of that time, and lack convenient ways of representing multiway selections (`switch()` statements etc).

At one time, complete programs were diagrammed using flowcharts. Such program flowcharts aren't that helpful, for the same reason that full call graphs aren't that helpful. There is too much detail in the diagram, so it doesn't effectively convey a program's structure.

It is possible to "flowchart" individual functions, an example in flowcharting style is shown in Figure 15.4. Most people find pseudo-code outlines easier to understand.

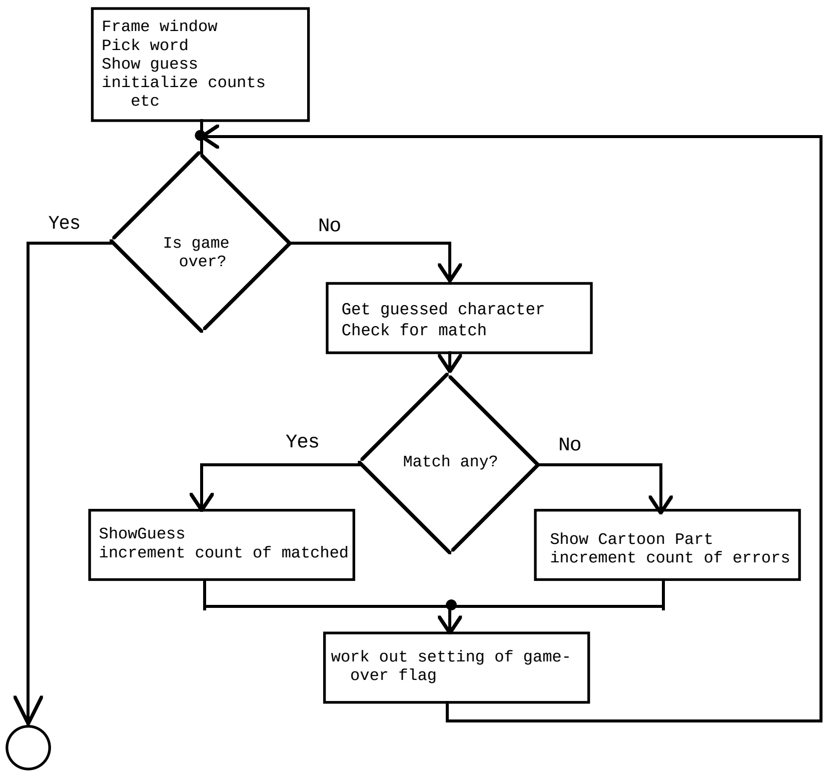

Figure 15.4 "Flowchart" representation, an alternative to a pseudo-code outline of a function.

On the whole, diagrams aren't that helpful. The lists of functions, the pseudo-code outlines etc are easier to understand and so it isn't worth trying to generate diagrammatic representations of the same information.

However, you may have access to a CASE ("Computer Assisted Software Engineering") program. These programs have a user interface somewhat similar to a drawing package. There is a palette of tools that you can use to place statement sequences, conditional tests, iterative constructs etc. Dialogs allow you to enter details of the operations involved.

The CASE tool can generate the final function prototypes, and the code for their implementation, from the information that you provide in these dialogs. If you use such a program, you automatically get a diagrammatic representation of your program along with the code.