28 Design and documentation: 2

The examples given in earlier chapters have in a rather informal way illustrated a design style that could be termed "object-based design".

At the most general level, there are similarities to the "top down functional decomposition" approach discussed in Chapter 15. A design process involves:

- *decomposition*, i.e. breaking a problem into smaller subparts that can be dealt with largely independently;

and

- *iteration*, a problem gets worked through many times at increasing levels of detail; provisional decisions get made, are tested through prototyping, and subsequently may be revised.

The most significant difference between object-based design and functional decomposition is in the initial focus for the decomposition process.

A focus on function

Top-down functional decomposition is an effective approach when the program has the basic form: input some data, compute some "function" of the data, output the result. In addition, the data must be simple in form. In such cases, the "function" that must be computed using the data serves as an obvious focus for the design process.

There are still many scientific, statistical, and business applications that have this relatively simple form. Examples are things like "compute the neutron flux in this reactor", or "calculate our total wage and tax bills given these employee time sheets". The function that must be computed for the data may be complex, but the overall program structure is simple.

"Multi-functional event handler programs"

Other programs don't fit this pattern. Many are "event handlers". They have some source of "events". These "events" may result from a user working interactively with keyed commands, menu-selections, or mouse actions. In other cases, the "events" may

come through external network connections reflecting actions by collaborating workers or autonomous devices. In simulation programs, the "events" may be created by the program's own logic.

"Events" represent requests for particular actions to be performed, particular functions to be executed. The program has many functions. The ones that are performed, and the sequence in which they are performed depends on the events received. The event patterns, and hence the function calls, differ in every run of the program.

Events often involve the creation (or deletion) of data elements. These data elements are of many different types. Their lifetimes vary. They are interrelated in complex ways. Frequently, simple individual data elements are linked together through pointers to build up models of more elaborate data structures.

These programs may still involve complex calculations. For example, you could have a "Computer Aided Design" (CAD) engineering program for designing nuclear reactors. Such a program would involve a sophisticated interactive component (a little like one of the better draw programs that you can get for personal computers), components for display of three dimensional structures, and – a complex function to calculate neutron fluxes. But the calculation is only one minor aspect of the overall program.

It is possible to design these programs using top-down functional decomposition. Though possible, such an approach to design is frequently problematical. There is no obvious starting point, no top from which to decompose downwards. You can identify the different main functions, e.g. edit, 3-d model, calculate, and you can decompose each of these. But then you end up with separate groups of functions that have to communicate through some shared global data. It is very easy for inconsistencies to arise where the different groups of functions embody different assumptions about the shared data.

A focus on data

Object-based programming is a design approach that helps deal with these more complex programs. The focus switches to the data.

The data will involve composites and simple data elements. For example, a CAD program has an overall structure (the engineering component being built) that is a composite made up from many individual parts of different types. Although there are different kinds of individual part (pipe, bracket, rod, …), all have data such as dimensions, mass, material type, and all can do things like draw themselves on a screen. The individual parts can look after their own data, the composite structure can deal with things like organizing a display (identify those parts that would be visible, tell them to draw themselves).

Inherently, data objects provide a basis for problem decomposition. If you can identify a group of data values that belong together, and a set of transformations that may be applied to those data, you have found a separable component for the overall program. You will be able to abstract out that component out and think about in isolation. You can design a class that describes the data owned and the things that can be done.

Sometimes, the things that an object must do are complex (e.g. the `Manager` object's application of the scheduling rules in the Supermarket example). In such cases, you can adopt a "top-down functional decomposition approach" because you are in the same situation as before – there is one clearly defined function to perform and the data are relatively simple (the data needed will all be represented as data members of the object performing the action).

You can code and test a class that defines any one type of data and the related functionality. Then you can return to the whole problem. But now the problem has been simplified because some details are now packaged away into the already built component.

When you return to the whole problem, you try to characterize the workings program in terms of interactions amongst example objects: e.g. "the structure will ask each part in its components list to draw itself".

If you design using top-down functional decomposition, you tend to see each problem as unique. If take an object based approach, you are more likely to see commonalities.

Opportunity for reuse

As just noted, most object-based programs seem to have "composite" structures that group separate components. The mechanisms for "grouping" are independent of the specific application. Consequently, you can almost always find opportunities for reusing standard components like lists, dynamic arrays, priority queues. You don't need to create a special purpose storage structure; you reuse a standard class.

An object-based approach to design has two major benefits: i) a cleaner, more effective decomposition of a complex problem, and ii) the opportunity to reuse components.

The use of abstract classes and inheritance, "object oriented design", brings further benefits. These have been hinted at in the Supermarket example, and in the discussion above regarding the CAD program and the parts that it manipulates. These design benefits are explored a little more in Part V.

28.1 OBJECT-BASED DESIGN

In Chapter 15, on "top-down functional decomposition", it was suggested that you could begin with a phrase or one sentence summary that defines the program. That doesn't help too much for something like the Supermarket, the InfoStore, or even the RefCards example. One sentence summaries don't say much.

"The RefCards program allows a user to keep a collection of reference cards." What are "reference cards"? Does the user do anything apart from "keep" them ("keeping" doesn't sound very interesting)? Explanations as to what the program really does usually fill several pages.

So, where do you begin?

Beginning

You begin by trying to answer the questions:

- What are the objects?
- What do they own?
- What do they do?

You start by thinking about prototypical objects, not the classes and certainly not abstract class hierarchies.

Identifying prototypical objects

Some ideas for the prototypical objects can come from a detailed reading of the full specification of the program (the "underline the nouns" approach). As previously noted, there are problems with too literally underlining the nouns; you may end modelling the world in too much detail. But it is a starting point for getting ideas as to things that might be among the more important objects – thus, you can pick up the need for `Customers` and `Checkouts` from the description of the Supermarket problem.

Usually, you will find that the program has a few objects that seem to be in for the duration, e.g. the `UserInteraction` and `CardCollection` objects in the RefCards program, or the `Shop`, `Manager`, and `Door` objects in the Supermarket example. In addition there are other objects that may be transient (e.g. the `Customers`). An important aspect of the design will be keeping track of when objects get created and destroyed.

Scenarios-1

Once you have formed at least some idea as to the objects that might be present in the executing program, it is worthwhile focussing on "events" that the program deals with and the objects that are involved. This process helps clarify what each kind of object owns and does, and also begins to establish the patterns of communication amongst classes.

You make up scenarios for each of the important "events" handled by the program. They should include the scenarios that show how objects get created and destroyed.

You must then compare the scenarios to check that you are treating the objects in a consistent manner. At the same time, you make up lists of what objects are asked to do and what data values you think that they should own.

Products of the first step

Once you have seen the ways that your putative objects behave in these scenarios you compose your initial class descriptions. These will include:

- class name, e.g. Shop
- data owned: e.g. "several histograms, some Lists to store Checkouts, a timer value, …"
- responsibilities: "adds a checkout (requested by Manager), notes when a checkout becomes idle (Checkout), finds a Checkout for a Customer (Customer), reports the time (various client classes), …"
- uses: (summary of requests made to instances of other classes), e.g. "Run() (all Activity subclasses), Checkout::AddCustomer(), …"

The responsibilities of the classes are all the things that you have seen being asked of prototypical instances in the scenarios that you have composed. It is often worthwhile noting the classes of client objects that use the functions of a class. In addition, you should note all the requests made to instances of other classes.

"Collaborators"

The pattern of requests made to and by an instance of a class identify its collaborators. If two objects are to collaborate, they have to have pointers to one another (e.g. the `Shop` had a `Manager*` pointer, and the collaborating `Manager` had a `Shop*` pointer). These pointers must get set before they need to be used.

Setting up the pointers linking collaborators hasn't been a problem in the examples presented so far. In more complex programs, the establishment of collaboration links can become an issue. Problems tend to be associated with situations where instances of one class can be created in different ways (e.g. read from a file or interactive command from a user). In one situation, it may be obvious that a link should be set to a collaborator. In the other situation, it may not be so obvious, and the link setting step may be forgotten. Forgetting links results in problems where a program seems to work intermittently.

The highlighting of collaborations in the early design stage can act as a reminder so that later on, when considering how instances of classes are created, you can remember to check that all required links are being set.

Isolable components

Sometimes you will get a class whose instances get asked to look after data and perform various tasks related to their data, but which don't themselves make requests to any other objects in the system. They act as "servers" rather than "clients" in all the "collaborations" in which they participate.

Such classes represent completely isolable components. They should be taken out of the main development. They can be implemented and tested in isolation. Then they can be reintroduced as "reusable" classes with the same standing as classes from standard libraries. The InfoStore example program provides an example; its `Vocab` class was isolable in this way.

Class hierarchies

You will get class hierarchies in two ways. Occasionally, the application problem will already have a hierarchy defined. The usual example quoted is a program that must manipulate "bank accounts". Now "bank accounts" are objects that do various things like accept debits and credits, report their balance, charge bank fees, and (sometimes) pay interest. A bank may have several different kinds of account, each with rather different rules regarding fees and interest payments. Here a hierarchy is obvious from the start. You have the abstract class "bank_account" which has pure virtual functions "DeductCharges()" and "AddInterest()". Then there are the various specialized subclasses ("loan_account", "savings", "checking", "checking_interest") that implement distinct versions of the virtual functions.

Other cases are more like the Supermarket example. There we had classes `Manager`, `Door`, `Checkout`, and `Customer` whose instances all had to behave "in the same way" so as to make it practical for the simulation system to use a single priority queue. This was handled by the introduction of an abstraction, class `Activity`, that became the base class for the other classes. Class `Activity` wasn't essential (the `Shop` could have used

four different priority queues); but its introduction greatly simplified design. The class hierarchy is certainly not one that you would have initially expected and doesn't reflect any "real world" relationship. (How many common features can you identify between "doors" and "customers"?)

Second step

Your initial classes are little more than "fuzzy blob" outlines. You have some idea as to the data owned and responsibilities but details will not have been defined. For example, you may have decided that "class Vocab owns the vocabulary and provides both fast lookup of words and listings of details", or that "class Manager handles the scheduling rules". You won't necessarily have decided the exact form of the data (hash-table or tree), you won't have all the data members (generally extra counters and flags get added to the data members when you get into more detail), and you certainly won't have much idea as to how the functions work and whether they necessitate auxiliary functions.

The next step in design is, considering the classes individually, to try to move from a "fuzzy blob" outline to something with a firm definition. You have to define the types of all data members (and get into issues like how data members should be initialized). Each member function has to be considered, possibly being decomposed into simpler auxiliary private member functions.

Outputs from design step

The output of this step should be the class declarations and lists of member functions like those illustrated in the various examples. Pseudo-code outlines should be provided for all the more complex member functions.

main()

The `main()` function is usually trivial: create the principle object, tell it to run.

Module structure

These programs are generally built from many separate files. The design process should also cover the module (file) structure and the "header dependencies". Details should be included with the rest of the design in the form of a diagram like that shown in 27.6.

Tests

As always, some thought has to be given to the testing of individual components and of the overall program.

28.2 DOCUMENTING A DESIGN

Diagrams are a much more important part of the documentation of the design of an object-based program than they were for the "top-down functional decomposition" programs.

Your documentation should include:

- "fuzzy" blob diagrams showing the classes and their principle roles (e.g. Figures 22.1 and 22.9);
- a hierarchy diagram (if needed); this could be defined in terms of the fuzzy blob classes (e.g. Figure 27.4) or the later design classes;
- scenarios for all important interactions among instances of classes;

- class "design diagrams" that summarize the data and function members of a class, (e.g. Figures 27.8 and 27.9);
- module structure;
- class declarations and member function summaries;
- pseudo-code outlines for complex functions.