20

Design pattern: multi-panel interactive systems

In our first example we will devise a design pattern which, in addition to illustrating some typical properties of the object-oriented method, provides an excellent opportunity to contrast it with other approaches, in particular top-down functional decomposition.

Because this example nicely captures on a small scale some of the principal properties of object-oriented software construction, I have often used it when requested to introduce an audience to the method in a few hours. By showing concretely (even to people who have had very little theoretical preparation) how one can proceed from a classical decomposition to an O-O view of things, and the benefits gained in this transformation, it serves as a remarkable pedagogical device. This chapter has been written so that it could play the same role for readers who have been directed to it by the reference they found in the “spoiler” chapter at the beginning of this book.

To facilitate their task, it has been made as self-contained as possible; this is why you will find a few repetitions with previous chapters, in particular a few short definitions of concepts which you already know inside out if you have been reading this book sequentially and carefully from the start.

20.1 MULTI-PANEL SYSTEMS

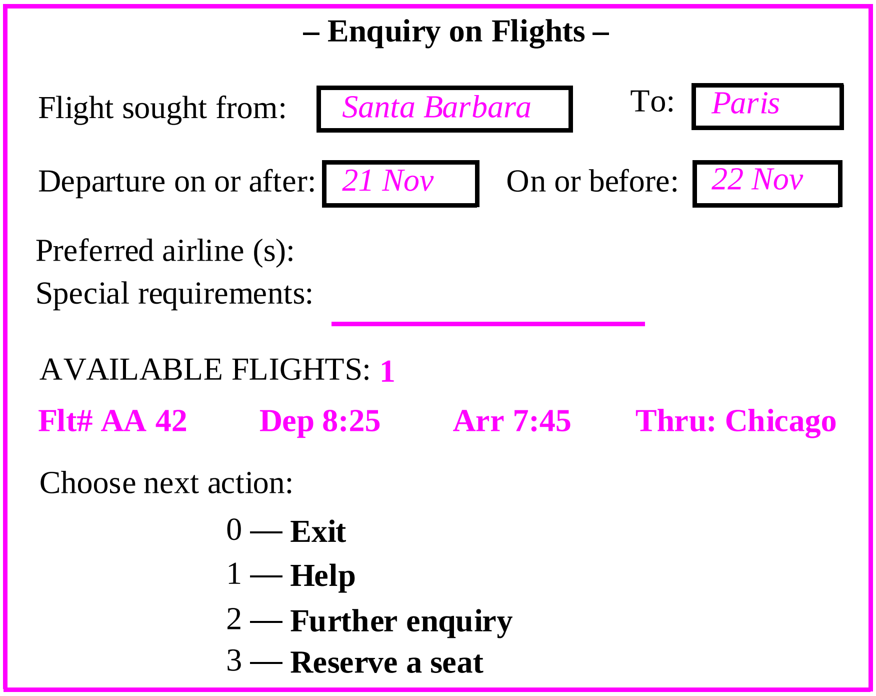
The problem is to write a system covering a general type of interactive system, common in business data processing, in which users are guided at each step of a session by a full-screen panel, with predefined transitions between the available panels.

The general pattern is simple and well defined. Each session goes through a certain number of *states*. In each state, a certain panel is displayed, showing questions to the user. The user will fill in the required answer; this answer will be checked for consistency (and questions asked again until an acceptable answer is found); then the answer will be processed in some fashion; for example the system will update a database. A part of the user’s answer will be a choice for the next step to perform, which the system will interpret as a transition to another state, where the same process will be applied again.

A typical example would be an airline reservation system, where the states might represent such steps of the processing as User Identification, Enquiry on Flights (for a certain itinerary on a certain date), Enquiry on Seats (for a certain flight) and Reservation.

A typical panel, for the Enquiry on Flights state, might look like the following (only intended, however, to illustrate the ideas, and making no claim of realism or good ergonomic design). The screen is shown towards the end of a step; items in *color italics* are the user's answers, and items in **bold color** show an answer displayed by the system.

A panel



– Enquiry on Flights –

Flight sought from: *Santa Barbara* To: *Paris*

Departure on or after: *21 Nov* On or before: *22 Nov*

Preferred airline (s):

Special requirements:

AVAILABLE FLIGHTS: **1**

Flt# AA 42 **Dep 8:25** **Arr 7:45** **Thru: Chicago**

Choose next action:

0 — **Exit**

1 — **Help**

2 — **Further enquiry**

3 — **Reserve a seat**

The session begins in an initial state, and ends whenever it reaches a final state. We can represent the overall structure by a transition graph showing the possible states and the transitions between them. The edges of the graph are labeled by integers corresponding to the possible user choices for the next step at the end of a state. At the top of the facing page is a graph for a simple airline reservation system.

The figure also include state numbers, for use later in the discussion.

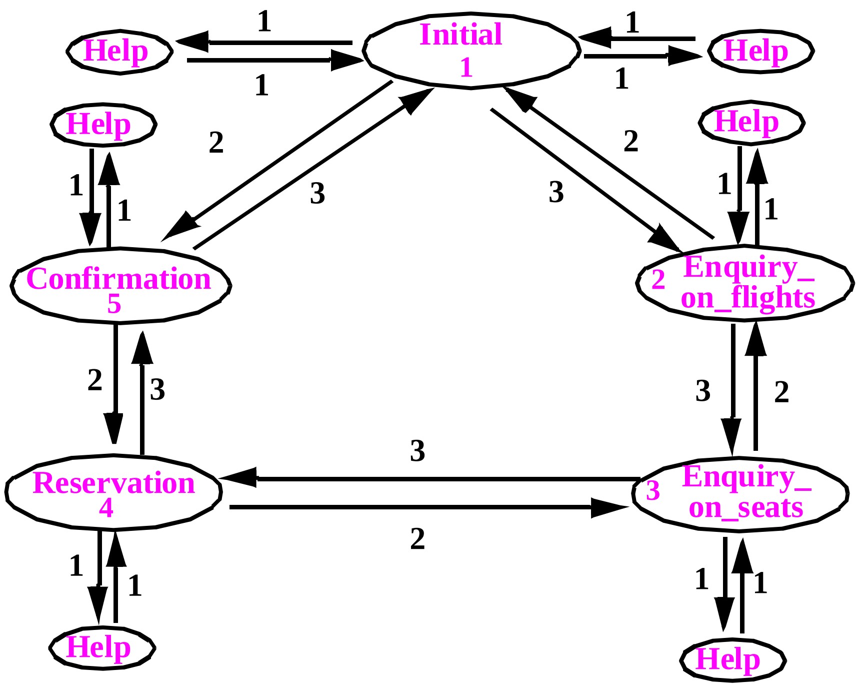
The problem is to come up with a design and implementation for such applications, achieving as much generality and flexibility as possible. In particular:

G1 • The graph may be large. It is not uncommon to see applications with several hundred states and correspondingly many transitions.

G2 • The structure is subject to change. The designers are unlikely to foresee all the possible states and transitions. As users start exercising the system, they will come up with requests for changes and additions.

G3 • Nothing in the given scheme is specific to the choice of application: the airline reservation mini-system is just a working example. If your company needs a number of such systems, either for its own purposes or (in a software house) for various customers, it will be a big benefit to define a general design or, better yet, a set of modules that you can reuse from application to application.

A transition diagram



20.2 A SIMPLE-MINDED ATTEMPT

Let us begin with a straightforward, unsophisticated program scheme. This version is made of a number of blocks, one for each state of the system: $B_{Enquiry}$, $B_{Reservation}$, $B_{Cancellation}$ etc. A typical block (expressed in an ad hoc notation, not the object-oriented notation of this book although it retains some of its syntactic conventions) looks like this:

```
B_Enquiry:
    "Display Enquiry on flights panel"
    repeat
        "Read user's answers and choice C for the next step"
        if "Error in answer" then "Output appropriate message" end
    until not error in answer end
    "Process answer"
    case C in
        C_0: goto Exit,
        C_1: goto B_Help,
        C_2: goto B_Reservation,
        …
    end
```

and similarly for each state.

This structure has something to speak for it: it is not hard to devise, and it will do the job. But from a software engineering viewpoint it leaves much to be desired.

The most obvious criticism is the presence of **goto** instructions (implementing conditional jumps similar to the **switch** of C and the "Computed Goto" of Fortran), giving the control structure that unmistakable "spaghetti bowl" look.

But the **goto**s are the symptom, not the real flaw. We have taken the superficial structure of the problem — the current form of the transition diagram — and hardwired it into the algorithm; the branching structure of the program is an exact reflection of the structure of the transition graph. This makes the software's design vulnerable to any of the simple and common changes cited above: any time someone asks us to add a state or change a transition, we will have to change the system's central control structure. And we can forget, of course, any hope of reusability across applications (goal G3 in the above list), as the control structure would have to cover all applications.

This example is a sobering reminder that we should never get carried away when we hear about the benefits of "modeling the real world" or "deducing the system from the analysis of the reality". Depending on how you describe it, the real world can be simple or messy; a bad model will give bad software. What counts is not how close the software is to the real world, but how good the description is. More on this topic at the end of this chapter.

To obtain not just a system but a good system we must think a little harder.

20.3 A FUNCTIONAL, TOP-DOWN SOLUTION

Repeating on this particular example the evolution of the programming species as a whole, we will go from a low-level **goto**-based structure to a top-down, hierarchically organized solution, analyze its own limitations, and only then move on to an object-oriented version. The hierarchical solution belongs to a general style also known as "structured", although this term should be used with care.

For one thing, an O-O solution is certainly structured too, although more in the sense of "structured programming" as originally introduced in the seventies by Dijkstra and others than relative to the quite distinct notion of "structured design".

The transition function

The first step towards improving the solution is to get rid of the central role of the traversal algorithm in the software's structure. The transition diagram is just one property of the system and it has no reason to rule over everything else. Separating it from the rest of the algorithm will, if nothing else, rid us of the **goto** instructions. And we should also gain generality, since the transition diagram depends on the specific application, such as airline reservation, whereas its traversal may be described generically.

What is the transition diagram? Abstractly, it is a function *transition* taking two arguments, a state and a user choice, such that *transition* (*s*, *c*) is the state obtained when the user chooses *c* when leaving state *s*. Here the word "function" is used in its mathematical sense; at the software level we can choose to implement *transition* either by a function in the software sense (a routine returning a value) or by a data structure such as an array. For the moment we can afford to postpone the choice between these solutions and just rely on *transition* as an abstract notion.

In addition to the function *transition* we also need to designate one of the states, say state *initial*, as the place where all sessions start, and to designate one or more states as final through a boolean-valued function *is_final*. Again this is a function in the mathematical sense, regardless of its eventual implementation.

We can picture the *transition* function in tabular form, with rows representing states and columns representing choices, as shown below.

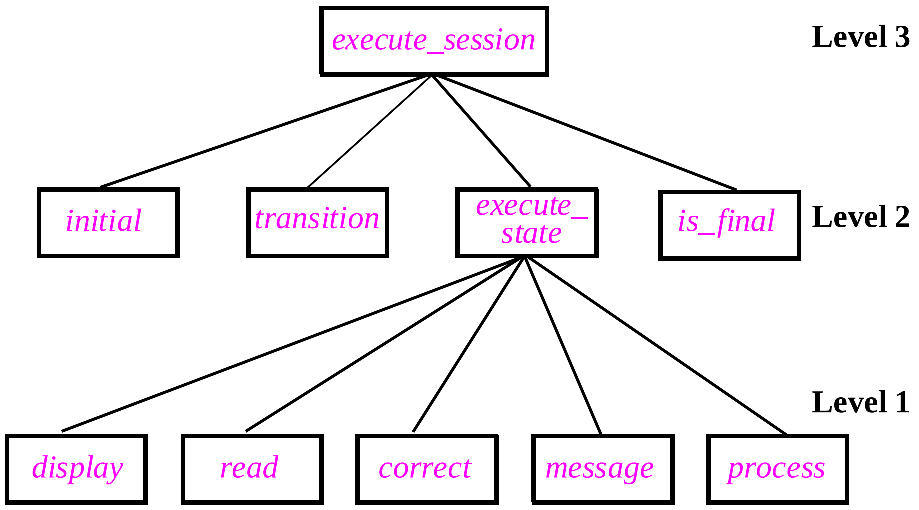
Conventions used in this table: there is just one *Help* state, **0**, with a special transition *Return* which goes back to the state from which *Help* was reached, and just one final state, **–1**. These conventions will not be necessary for the rest of the discussion but help keep the table simple.

A transition table

Choice → **↓ State**	**0**	**1**	**2**	**3**
1 (*Initial*)	**–1**	**0**	**5**	**2**
2 (*Flights*)		**0**	**1**	**3**
3 (*Seats*)		**0**	**2**	**4**
4 (*Reserv.*)		**0**	**3**	**5**
5 (*Confirm*)		**0**	**4**	**1**
0 (*Help*)		*Return*		
–1 (*Final*)				

The routine architecture

Following the traditional precepts of top-down decomposition, we choose a "top" (the main program) for our system. This should clearly be the routine *execute_session* that describes how to execute a complete interactive session.



Top-down functional decomposition

Immediately below (level 2) we will find the operations relative to states: definition of the initial and final states, transition structure, and *execute_state* which prescribes the actions to be executed in each state. Then at the lowest level (1) we will find the constituent operations of *execute_state*: display a screen and so on. Note how such a solution may be described, as well as anything object-oriented that we may later see, to “reflect the real world”: the structure of the software perfectly mirrors the structure of an application, which involves states, which involve elementary operations. Real-worldliness is not, in this example and many others, a significant difference between O-O and other approaches; what counts is *how* we model the world.

In writing *execute_session* let us try to make it as application-independent as possible. (The routine is again expressed in an ad hoc notation imitated from the O-O notation of the rest of this book. The **repeat** … **until** … loop is borrowed from Pascal.)

```
execute_session is
        -- Execute a complete session of the interactive system
    local
        state, choice: INTEGER
    do
        state := initial
        repeat
            execute_state (state, →next)
                -- Routine execute_state updates the value of next.

            state := transition (state, next)
        until is_final (state) end
    end
```

This is a typical transition diagram traversal algorithm. (The reader who has written a lexical analyzer will recognize the pattern.) At each stage we are in a state *state*, originally set to *initial*; the process terminates when *state* satisfies *is_final*. For a non-final state we execute *execute_state*, which takes the current state and returns the user's transition choice through its second argument *next*, which the function *transition* uses, together with *state*, to determine the next state.

The → notation is a temporary convention, used only for this particular procedure and for read below.

The technique using a procedure *execute_state* that changes the value of one of its arguments would never be appropriate in good O-O design, but here it is the most expedient. To signal it clearly, the notation flags an "out" argument such as *next* with an arrow →. Instead of a procedure which modifies an argument, C developers would make *execute_state* a side-effect-producing function called as *next* := *execute_state* (*state*); we will see that this practice is subject to criticism too.

Since *execute_state* does not show any information about any particular interactive application, you must fill in the application-specific properties appearing on level 2 in the figure: *transition* function; *initial* state; *is_final* predicate.

To complete the design, we must refine the *execute_state* routine describing the actions to be performed in each state. Its body is essentially an abstracted form of the contents of the successive blocks in the initial **goto**-based version:

```
execute_state (in s: INTEGER; out c: INTEGER) is
        -- Execute the actions associated with state s,
        -- returning into c the user's choice for the next state.
    local
        a: ANSWER; ok: BOOLEAN
    do
        repeat
            display (s)
            read (s, →a)
            ok := correct (s, a)
            if not ok then message (s, a) end
        until ok end
        process (s, a)
        c := next_choice (a)
    end
```

This assumes level 1 routines with the following roles:

- *display* (*s*) outputs the panel associated with state *s*.
- *read* (*s*, →*a*) reads into *a* the user's answer to the display panel of state *s*.
- *correct* (*s*, *a*) returns true if and only if *a* is an acceptable answer to the question displayed in state *s*; if so, *process* (*s*, *a*) processes answer *a*, for example by updating a database or displaying more information; if not, *message* (*s*, *a*) outputs the relevant error message.

The type *ANSWER* of the object representing the user's answer has not been refined further. A value *a* of that type globally represents the input entered by the user in a given state; it is assumed to include the user's choice for the next step, written *next_choice* (*a*). (*ANSWER* is in fact already very much like a class, even though the rest of the architecture is not object-oriented at all.)

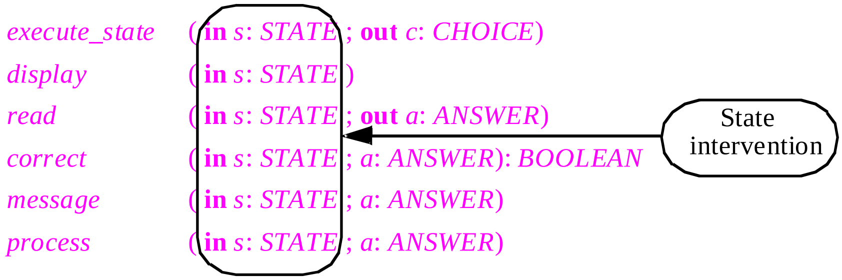
To obtain a working application, you will need to fill in the various level 1 features: *display*, *read*, *correct*, *message* and *process*.

20.4 A CRITIQUE OF THE SOLUTION

Have we now a satisfactory solution? Not quite. It is better than the first version, but still falls short of our goals of extendibility and reusability.

Statism

Although on the surface it seems we have been able to separate the generic from the application-specific, in reality the various modules are still tightly coupled with each other and with the choice of application. The main problem is the data transmission structure of the system. Consider the signatures (argument and result types) of the routines:



```
execute_state    (in s: STATE; out c: CHOICE)
display          (in s: STATE)
read             (in s: STATE; out a: ANSWER)
correct          (in s: STATE; a: ANSWER): BOOLEAN
message          (in s: STATE; a: ANSWER)
process          (in s: STATE; a: ANSWER)
```

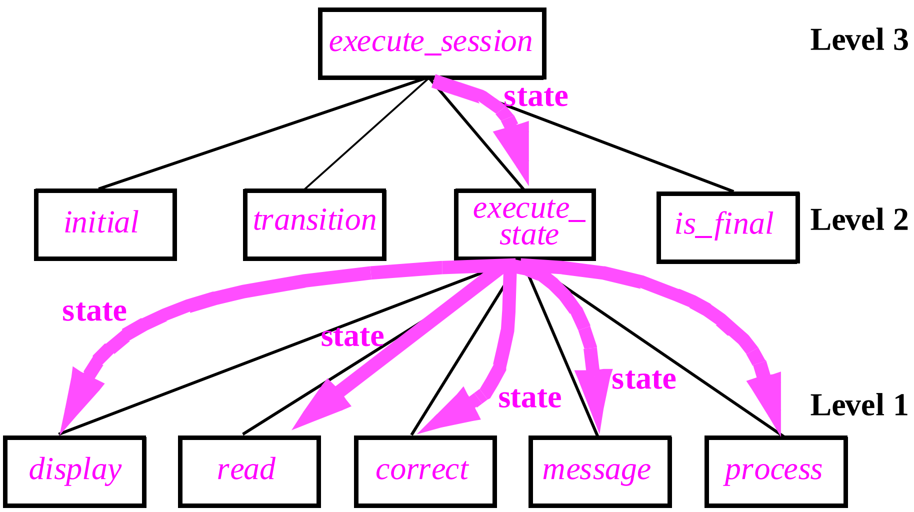
The observation (which sounds like an economist's lament) is that the role of the state is too pervasive. The current state appears under the name *s* as an argument in all the routines, coming from the top module *execute_session*, where it is known as *state*. So the hierarchical structure shown in the last figure, seemingly simple and manageable, is a lie, or more precisely a façade. Behind the formal elegance of the functional decomposition lies a jumble of data transmission. The true picture, shown at the top of the facing page, must involve the data.

The architectural figure is on page 680.

The background for object technology, as presented at the beginning of this book, is the battle between the *function* and *data* (object) aspects of software systems for control of the architecture. In non-O-O approaches, the functions rule unopposed over the data; but then the data take their revenge.

The revenge comes in the form of sabotage. By attacking the very foundations of the architecture, the data make the system impervious to change — until, like a government unable to handle its *perestroika*, it will crumble under its own weight.

The flow of data



In this example the subversion of the structure comes in particular from the need to discriminate on states. All the level 1 routines must perform different actions depending on *s*: to display the panel for a certain state; to read and interpret a user answer (made of a number of input fields, different for each state); to determine whether the answer is correct; to output the proper error message; to process a correct answer — you must know the state. The routines will perform a discrimination of the form

```
inspect
    s
when Initial then
    …
when Enquiry_on_flights then
    …
…
end
```

This means long and complex control structures and, worse yet, a fragile system: any addition of a state will require changes throughout the structure. This is a typical case of unbridled knowledge distribution: far too many modules of the system rely on a piece of information — the list of all possible states — which is subject to change.

The situation is in fact even worse than it appears if we are hoping for general reusable solutions. There is an extra implicit argument in all the routines considered so far: the *application* — airline reservation or anything else we are building. So to make routines such as *display* truly general we would have to let them know about all states of all possible applications in a given computing environment! Function *transition* would similarly contain the transition graph for all applications. This is of course unrealistic.

20.5 AN OBJECT-ORIENTED ARCHITECTURE

The very deficiencies of top-down functional decomposition point to what we must do to obtain a good object-oriented version.

The law of inversion

What went wrong? Too much data transmission in a software architecture usually signals a flaw in the design. The remedy, which leads directly to object-oriented design, may be expressed by the following design rule:

Law of inversion

If your routines exchange too many data, put your routines in your data.

Instead of building modules around operations (such as *execute_session* and *execute_state*) and distributing the data structures between the resulting routines, with all the unpleasant consequences that we have seen, object-oriented design does the reverse: it uses the most important data types as the basis for modularization, attaching each routine to the data type to which it relates most closely. When objects take over, their former masters, the functions, become their vassals.

The law of inversion is the key to obtaining an object-oriented design from a classical functional (procedural) decomposition, as in this chapter. Such a need arises in cases of *reverse-engineering* an existing non-O-O system to make it more maintainable and prepare its evolution; it is also frequent in teams that are new to object-oriented design and think “functional” first.

It is of course best to design in an object-oriented fashion from the beginning; then no inversion is needed. But the law of inversion is useful beyond cases of reverse-engineering and novice developers. Even someone who has been exposed to the principles of object-oriented software construction may come up with an initial design that has pockets of functional decomposition in an object landscape. Analyzing data transmission is a good way to detect and correct such design flaws. If you see — even in a structure intended as O-O — a data transmission pattern similar to what happens with states in the example of this chapter, it should catch your attention. Probing further will in most cases lead you to the discovery of a data abstraction that has not received its proper due in the software’s architecture.

State as a class

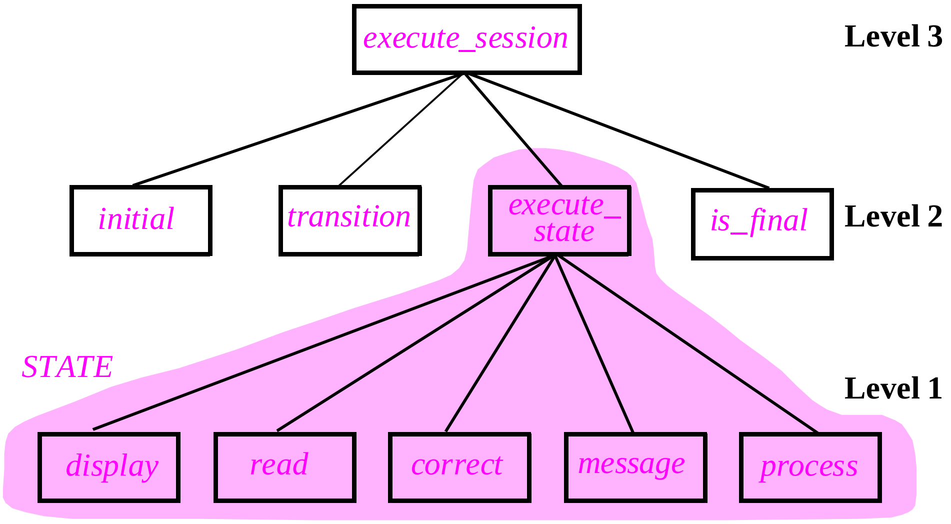
The “state” example is typical. Such a data type, appearing so pervasively in the data transmissions between routines, is a prime candidate for serving as one of the modular components of an object-oriented architecture, which must be based on classes (abstractly described data types).

The notion of state was important in the original problem statement, but in the functional architecture that importance was lost: the state was just represented by a variable, passed from routine to routine as if it were some kind of lowlife. We have seen how it avenged itself. Now we are ready to give it the status it deserves. *STATE* should be a class, one of the principals in the structure of our new object-oriented system.

In that class we will find all the operations that characterize a state: displaying the corresponding screen (*display*), analyzing a user's answer (*read*), checking the answer (*correct*), producing an error message for an incorrect answer (*message*), processing a correct answer (*process*). We must also include *execute_state*, expressing the sequence of actions to be performed whenever the session reaches a given state; since the original name would be over-qualifying in a class called *STATE*, we can replace it by just *execute*.

Starting from the original top-down functional decomposition picture, we can highlight the set of routines that should be handed over to *STATE*:

STATE ***features***



The class will have the following form:

… **class** *STATE* **feature**
input: *ANSWER*
choice: *INTEGER*
execute **is do** … **end**
display **is** …
read **is** …
correct: *BOOLEAN* **is** …
message **is** …
process **is** …
end

Features *input* and *choice* are attributes; the others are routines. Compared to their counterparts in the functional decomposition, the routines have lost their explicit state arguments, although the state will reappear in calls made by clients, such as *s*.*execute*.

In the previous approach, *execute* (formerly *execute_state*) returned the user's choice for the next step. But such a style violates principles of good design. It is preferable to treat *execute* as a command, whose execution determines the result of the query "what choice did the user make in the last state?", available through the attribute *choice*. Similarly, the *ANSWER* argument to the level 1 routines is now replaced by the secret attribute *input*. The reason is information hiding: client code does not need to look at answers except through the interface provided by the exported features.

See "SIDE EFFECTS IN FUNCTIONS", 23.1, page 748.

Inheritance and deferred classes

Class *STATE* does not describe a particular state, but the general notion of state. Procedure *execute* is the same for all states, but the other routines are state-specific.

Inheritance and deferred classes ideally address such situations. At the *STATE* level, we know the procedure *execute* in full detail and the attributes. We also know the existence of the level 1 routines (*display* etc.) and their specifications, but not their implementations. These routines should be deferred; class *STATE*, which describes a set of variants, rather than a fully spelled out abstraction, is itself a deferred class. This gives:

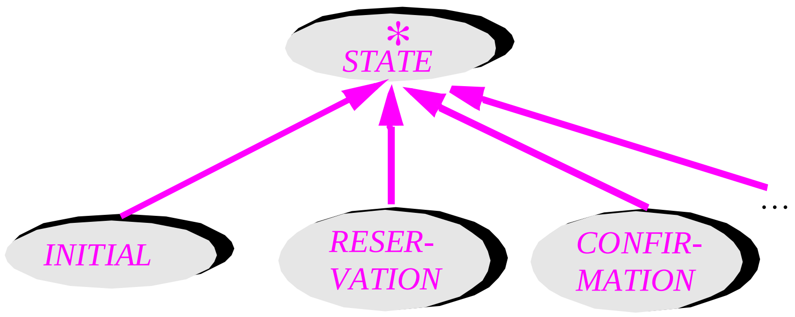
```
indexing
    description: "States for interactive panel-driven applications"
deferred class
    STATE
feature -- Access
    choice: INTEGER
            -- User's choice for next step

    input: ANSWER
            -- User's answer to questions asked in this state.
feature -- Status report
    correct: BOOLEAN is
                -- Is input a correct answer?
            deferred
            end
feature -- Basic operations
    display is
                -- Display panel associated with current state.
            deferred
            end
```

It is easy to remove the test from within the loop for better efficiency.

```
    execute is
            -- Execute actions associated with current state
            -- and set choice to denote user's choice for next state.
        local
            ok: BOOLEAN
        do
            from ok := False until ok loop
                display; read; ok := correct
                if not ok then message end
            end
            process
        ensure
            ok
        end

    message is
            -- Output error message corresponding to input.
        require
            not correct
        deferred
        end

    read is
            -- Obtain user's answer into input and choice into next_choice.
        deferred
        end

    process is
            -- Process input.
        require
            correct
        deferred
        end
end -- class STATE
```

To describe a specific state you will introduce descendants of *STATE* providing effectings (implementations) of the deferred features:

State class hierarchy



An example would look like:

```
class ENQUIRY_ON_FLIGHTS inherit
    STATE
feature
    display is
        do
            … Specific display procedure …
        end
    … And similarly for read, correct, message and process …
end -- class ENQUIRY_ON_FLIGHTS
```

This architecture separates, at the exact grain of detail required, elements common to all states and elements specific to individual states. The common elements, such as procedure *execute*, are concentrated in *STATE* and do not need to be redeclared in descendants such as *ENQUIRY_ON_FLIGHTS*. The Open-Closed principle is satisfied: *STATE* is closed in that it is a well-defined, compilable unit; but it is also open, since you can add any number of descendants at any time.

STATE is typical of **behavior classes** — deferred classes capturing the common behavior of a large number of possible objects, implementing what is fully known at the most general level (*execute*) in terms of what depends on each variant. Inheritance and the deferred mechanism are essential to capture such behavior in a self-contained reusable component.

"Don't call us, we'll call you", page 505.

Describing a complete system

To complete the design we must still take care of managing a session. In the functional decomposition this was the task of procedure *execute_session*, the main program. But now we know better. As discussed in an earlier chapter, the "topmost function of a system" as posited in the top-down method is mythical. A large software system performs many equally important functions. Here again, the abstract data type approach is more appropriate; it considers the system, taken as a whole, as a set of abstract objects capable of rendering a certain number of services.

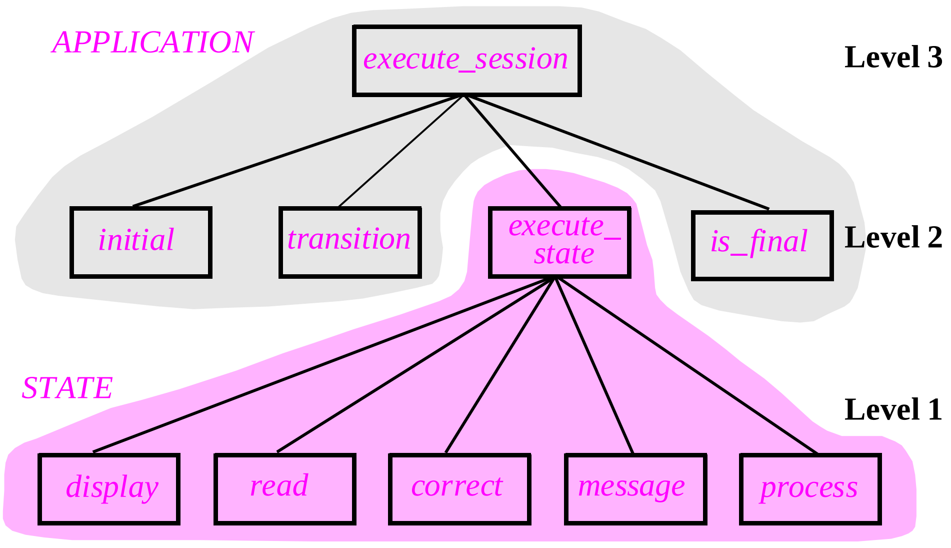
"Finding the top", page 107.

We have captured one key abstraction: *STATE* (along with *ANSWER*). What abstraction is our design still missing? Central in the understanding of the problem is the notion of *APPLICATION*, describing specific interactive systems such as the airline reservation system. This will yield a new class.

It turns out that the remaining components of the functional decomposition, shown in the figure, are all features of an application and will find their true calling as features of class *APPLICATION*:

- *execute_session*, describing how to execute an application. Here the name will be simplified to *execute* since the enclosing class provides qualification enough (and there is no possible confusion with *execute* of *STATE*).

STATE **and** *APPLICATION* ***features***



- *initial* and *is_final*, indicating which states have special status in an application. Note that it is proper to have these features in *APPLICATION* rather than *STATE* since they describe properties of applications rather than states: a state is not initial or final per se, but only with respect to an application. (If we reuse states between applications, a state may well be final in a certain application but not in another.)
- *transition* to describe the transition between states in the application.

The components of the functional decomposition have all found a place as features of the classes in the O-O decomposition — some in *STATE*, some in *APPLICATION*. This should not surprise us. Object technology, as has been repeatedly emphasized in this book, is before anything else an *architectural* mechanism, primarily affecting how we organize software elements into coherent structures. The elements themselves may be, at the lowest level, the same ones that you would find in a non-O-O solution, or at least similar (data abstraction, information hiding, assertions, inheritance, polymorphism and dynamic binding help make them more simple, general and powerful).

A panel-driven system of the kind studied in this chapter will always need to have operations for traversing the application graph (*execute_session*, now *execute*), reading user input (*read*), detecting final states (*is_final*). Deep down in the structure, then, we will find some of the same building blocks regardless of the method. What changes is how you group them to produce a modular architecture.

Of course we do not need to limit ourselves to features that come from the earlier solution. What for the functional decomposition was the end of the process — building *execute* for applications and all the other mechanisms that it needs — is now just a beginning. There are many more things we may want to do on an application:

- Add a new state.
- Add a new transition.
- Build an application (by repeated application of the preceding two operations).
- Remove a state, a transition.
- Store the complete application, its states and transitions, into a database.
- Simulate the application (for example on a line-oriented display, or with stubs replacing the routines of class *STATE*, to check the transitions only).
- Monitor usage of the application.

All these operations, and others, will yield features of class *APPLICATION*. They are no less and no more important than our former "main program", procedure *execute*, now just one of the features of the class, *inter pares* but not even *primus*. By renouncing the notion of top, we make room for evolution and reuse.

The application class

To finish class *APPLICATION* here are a few possible implementation decisions:

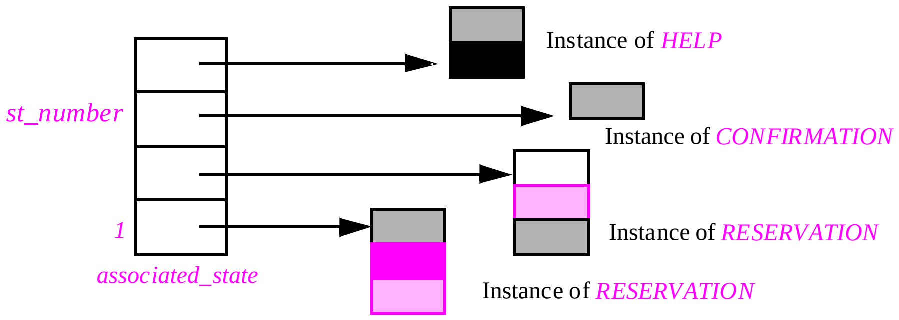
- Number states 1 to *n* for the application. Note that these numbers are not absolute properties of the states, but only relative to a certain application; so there is no "state number" attribute in class *STATE*. Instead, a one-dimensional array *associated_state*, an attribute of *APPLICATION*, yields the state associated with a given number.
- Represent the *transition* function by another attribute, a two-dimensional array of size $n \times m$, where m is the number of possible exit choices.
- The number of the initial state is kept in the attribute *initial* and set by the routine *choose_initial*. For final states we can use the convention that a transition to pseudo-state 0 denotes session termination.
- The creation procedure of *APPLICATION* uses the creation procedures of the library classes *ARRAY* and *ARRAY2*. The latter describes two-dimensional classes and is patterned after *ARRAY*; its creation procedure *make* takes four arguments, as in *!! a.make (1, 25, 1, 10)*, and its *item* and *put* routines use two indices, as in *a.put (x, 1, 2)*. The bounds of a two-dimensional array *a* are *a.lower1* etc.

Here is the class resulting from these decisions:

```
indexing
    description: "Interactive panel-driven applications"
class APPLICATION creation
    make
    feature -- Initialization
        make (n, m: INTEGER) is
                -- Allocate application with n states and m possible choices.
            do
                !! transition.make (1, n, 1, m)
                !! associated_state.make (1, n)
            end
    feature -- Access
        initial: INTEGER
                -- Initial state's number
    feature -- Basic operations
        execute is
                -- Perform a user session
            local
                st: STATE; st_number: INTEGER
            do
                from
                    st_number := initial
                invariant
                    0 <= st_number; st_number <= n
                until st_number = 0 loop
                    st := associated_state.item (st_number)
                    st.execute
                        -- This refers of course to the execute procedure of STATE
                        -- (see next page for comments on this key instruction).
                    st_number := transition.item (st_number, st.choice)
                end
            end
    feature -- Element change
        put_state (st: STATE; sn: INTEGER) is
                -- Enter state st with index sn.
            require
                1 <= sn; sn <= associated_state.upper
            do
                associated_state.put (st, sn)
            end
        choose_initial (sn: INTEGER) is
                    -- Define state number sn as the initial state.
                require
                    1 <= sn; sn <= associated_state.upper
                do
                    initial := sn
                end

        put_transition (source, target, label: INTEGER) is
                    -- Enter transition labeled label
                    -- from state number source to state number target.
                require
                    1 <= source; source <= associated_state.upper
                    0 <= target; target <= associated_state.upper
                    1 <= label; label <= transition.upper2
                do
                    transition.put (source, label, target)
                end
    feature {NONE} -- Implementation
        transition: ARRAY2 [STATE]
        associated_state: ARRAY [STATE]
        … Other features …
    invariant
        transition.upper1 = associated_state.upper
    end -- class APPLICATION
```

Note how simply and elegantly the highlighted call on the preceding page, *st.execute*, captures some of the problem's essential semantics. The feature called is *execute* from *STATE*; although effective because it describes a known general behavior, *execute* relies on deferred features *read*, *message*, *correct*, *display*, *process*, deferred at the level of *STATE* and effected only in its proper descendants such as *RESERVATION*. When we place the call *st.execute* in *APPLICATION*'s own *execute*, we have no idea what kind of state *st* denotes — although we do know that it is a state (this is the benefit of static typing). To come to life, the instruction needs the machinery of dynamic binding: when *st* becomes attached at run time to a state object of a particular kind, say *RESERVATION*, calls to *read*, *message* and consorts will automatically trigger the right version.

The value of *st* is obtained from *associated_state*, a **polymorphic data structure** which may contain objects of different types, all conforming to *STATE*. Whatever we find at the current index *st_number* will determine the next state operations.


Instance of *HELP*
st_number
Instance of *CONFIRMATION*
Instance of *RESERVATION*
1
associated_state
Instance of *RESERVATION*

Here is how you build an interactive application. The application will be represented by an entity, say *air_reservation*, declared of type *APPLICATION*. You must create the corresponding object:

!! *air_reservation*.*make* (*number_of_states*, *number_of_possible_choices*)

You will separately define and create the application's states as entities of descendant types of *STATE*, either new or reused from a state library. You assign to each state *s* a number *i* for the application:

air_reservation.*put_state* (*s*, *i*).

You choose one of the states, say the state numbered i_0, as initial:

air_reservation.*choose_initial* (i_0)

To set up a transition from state number *sn* to state number *tn*, with label *l*, you use

air_reservation.*enter_transition* (*sn*, *tn*, *l*)

This includes exit transitions, for which *tn* is 0 (the default). You may now execute the application:

air_reservation.*execute_session*.

During system evolution you may at any time use the same routines to add a new state or a new transition.

It is of course possible to extend class *APPLICATION*, either by changing it or by adding descendants, to accommodate more features such as deletion, simulation, or any of the others mentioned in the course of the presentation.

20.6 DISCUSSION

This example provides a striking picture of the differences between object-oriented software construction and earlier approaches. It shows in particular the benefits of getting rid of the notion of main program. By focusing on the data abstractions and forgetting, for as long as possible, what is "the" main function of the system, we obtain a structure that is much more likely to lend itself gracefully to future changes and to reuse across many different variants.

This equalizing effect is one of the characteristic properties of the method. It takes some discipline to apply it consistently, since it means resisting the constant temptation to ask: "What does the system do?". This is one of the skills that sets the true object-oriented professional from people who (although they may have been using O-O techniques and an O-O language for a while) have not yet digested the method, and will still produce functional architectures behind an object façade.

We have also seen a heuristic that is often useful to identify key abstractions in an object-oriented (to "find the objects", or rather the classes, the topic of a subsequent chapter): analyzing data transmissions and being on the lookout for notions that show up in communications between numerous components of a system. Often this is an indication that the structure should be turned upside down, the routines becoming attached to the data abstraction rather than the reverse.

A final lesson of this chapter is that you should be wary of attaching too much importance to the notion that object-oriented systems are directly deduced from the "real world". The modeling power of the method is indeed impressive, and it is pleasant to produce software architectures whose principal components directly reflect the abstractions of the external system being modeled. But there are many ways to model the real world, and not all of them will lead to a good system. Our first, **goto**-filled version was as close to the real world as the other two — closer actually, since it is directly patterned after the structure of the transition diagram, whereas the other two require introducing intermediate concepts. But it is a software engineering disaster.

In contrast, the object-oriented decomposition that we finally produced is good because the abstractions that it uses — *STATE*, *APPLICATION*, *ANSWER* — are clear, general, manageable, change-ready, and reusable across a broad application area. Although once you understand them they appear as real as anything else, to a newcomer they may appear less "natural" (that is to say, less close to an informal perception of the underlying reality) than the concepts used in the inferior solutions studied first.

To produce good software, what counts is not how close you are to someone's perception of the real world, but how good are the abstractions that you choose both to model the external systems and to structure your own software. This is indeed the very definition of object-oriented analysis, design and implementation, the task that you will have to execute well, day in and day out, to make your project succeed, and the skill that distinguishes object experts from object amateurs: *finding the right abstractions*.

20.7 BIBLIOGRAPHICAL NOTE

Variants of the example discussed in this chapter were used to illustrate object-oriented concepts in [M 1983] and [M 1987].