An O-O design example

A reservation panel

- Enquiry on Flights -
  Flight sought from: Santa Barbara  To: Zurich
  Departure on or after: 23 June  On or before: 24 June
  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

A first attempt

PEnquiry_on_flights:
output "enquiry on flights' screen
repeat
  read user's answers and his exit choice C
  if error in answer then
    output message
  end
  until no error in answer
  process answer
  inspect C
  when C=0 then
    output Exit
  when C=1 then
    output Help
  when C=2 then
    output Reservation
  when C=3 then
    output "(and similarly for each state)"
What's wrong with the previous scheme?

- Intricate branching structure ("spaghetti bowl").
- Extendibility problems: dialogue structure wired into program structure.

A functional, top-down solution

For more flexibility, represent the structure of the transition diagram by a function

\[ \text{transition}(i, k) \]

used to specify the transition diagram associated with any particular interactive application.

Function transition may be implemented as a data structure, for example a two-dimensional array.

The transition function

<table>
<thead>
<tr>
<th></th>
<th>0 (Initial)</th>
<th>1 (Help)</th>
<th>2 (Conf.)</th>
<th>3 (Reserv.)</th>
<th>4 (Seats)</th>
<th>5 (flights)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Exit</td>
<td></td>
<td></td>
<td>Exit</td>
<td></td>
<td>Exit</td>
</tr>
<tr>
<td>1</td>
<td>Exit</td>
<td>Return</td>
<td></td>
<td>Exit</td>
<td></td>
<td>Exit</td>
</tr>
<tr>
<td>2</td>
<td>Exit</td>
<td></td>
<td></td>
<td>Exit</td>
<td></td>
<td>Exit</td>
</tr>
<tr>
<td>3</td>
<td>Exit</td>
<td></td>
<td></td>
<td>Exit</td>
<td></td>
<td>Exit</td>
</tr>
<tr>
<td>4</td>
<td>Exit</td>
<td></td>
<td></td>
<td>Exit</td>
<td></td>
<td>Exit</td>
</tr>
<tr>
<td>5</td>
<td>Exit</td>
<td></td>
<td></td>
<td>Exit</td>
<td></td>
<td>Exit</td>
</tr>
</tbody>
</table>

The transition diagram
New system architecture

Procedure `execute_session` only defines graph traversal. Knows nothing about particular screens of a given application. Should be the same for all applications.

```plaintext
execute_session is
  local -- Execute full session
  do
    current_state, choice: INTEGER
    current_state := initial
    repeat
      choice := execute_state(current_state)
      current_state := transition(current_state, choice)
    until is_final(current_state)
  end
```

To describe an application

- Provide `transition` function
- Define `initial` state
- Define `is_final` function

Actions in a state

```plaintext
execute_state(current_state: INTEGER): INTEGER is
  -- Actions for current_state, returning user’s exit choice.
  local
    answer: ANSWER
    good: BOOLEAN
    choice: INTEGER
    do
      repeat
        display(current_state)
        [answer, choice] := read(current_state)
        good := correct(current_state, answer)
      until not good then
        message(current_state, answer)
      end
    until good
    end
    process(current_state, answer)
  return
  choice
```

Level 3
- `execute_session`

Level 2
- `initial`
- `transition`
- `execute_state`
- `is_final`

Level 1
- `display`
- `read`
- `correct`
- `message`
- `process`
Specification of the remaining routines

- \textit{display} (s) outputs the screen associated with state \( s \).
- \( [a, e] := \text{read} (s) \) reads into \( a \) the user's answer to the display screen of state \( s \), and into \( e \) the user's exit choice.
- \textit{correct} (s, a) returns true if and only if \( a \) is a correct answer for the question asked in state \( s \).
- If so, \textit{process} (s, a) processes answer \( a \).
- If not, \textit{message} (s, a) outputs the relevant error message.

Going object-oriented: The law of inversion

How amenable is this solution to change and adaptation?
- New transition?
- New state?
- New application?

Routine signatures:

- \textit{execute_state} (state \( \text{INTEGER} \): \( \text{INTEGER} \))
- \textit{display} (state \( \text{INTEGER} \))
- \textit{read} (state \( \text{INTEGER} \): \{\text{ANSWER, INTEGER}\})
- \textit{correct} (state \( \text{INTEGER} \), \( \text{ANSWER} \))
- \textit{message} (state \( \text{INTEGER} \), \( \text{ANSWER} \))
- \textit{process} (state \( \text{INTEGER} \), \( \text{ANSWER} \))
- \textit{is_final} (state \( \text{INTEGER} \))

Data transmission

All routines share the state as input argument. They must discriminate on that argument, e.g.:

\begin{verbatim}
  display (current_state \( \text{INTEGER} \)) is do
    inspect current_state
    when state then
      when state then
        when state then
          end
  end
\end{verbatim}

Consequences:
- Long and complicated routines.
- Must know about a possibly complex application.
- To change one transition, or add a state, need to change all.
The flow of control

Underlying reason why structure is so inflexible: Too much DATA TRANSMISSION.
Variable current_state is passed from execute_session (level 3) to all routines on level 2 and on to level 1
Worse: there's another implicit argument to all routines - application. Can't define execute_session, display, execute_state, ...
as library components, since each must know about all interactive applications that may use it.

The visible architecture

Level 3
execute_session

Level 2
initial transition execute_state is_final

display read correct message process

Level 1

The real story

Level 3
execute_session

Level 2
initial transition execute_state is_final
Level 1
state state state state state

display read correct message process

The law of inversion

The everywhere lurking state
- If your routines exchange data too much, put your routines into your data.
**Going O-O**

Use `STATE` as the basic abstract data type (yielding a class).

Among features of a state:

- The routines of level 1 (deferred in `STATE`)
- `execute_state`, as above but without `current_state` argument.

**Class STATE**

defined class

STATE

feature

choice: INTEGER

input: ANSWER

-- User's selection for next step

display is

(deferred

end)

read is

-- Show screen for this step.

-- Get user's answer and exit choice.

deferred ensure

input is Void

end

correct: BOOLEAN

-- Is input acceptable?

deferred end

**Grouping by data abstractions**

Level 3

```
execute_session
```

Level 2

```
initial transition execute_state is_final
```

STATE

Level 1

```
display read correct message process
```

**Class STATE**

*message is*

-- Display message for erroneous input.

```
require not correct deferred end
```

*process is*

-- Process correct input.

```
require correct deferred end
```
**Class STATE**

execute_state is
local
dolocal
fromuntil
loop
displayread
end
end
end
choice := input.choice

**To describe a state of an application**

Introduce new descendant of **STATE**:

class
  ENQUIRY_ON_FLIGHTS
inherit
  STATE
feature
  display is do ... end
  read is do ... end
  correct BOOLEAN is do ... end
  message is do ... end
  process is do ... end
end

**Class structure**

```
* STATE
  INITIAL
  RESERVATION
  CONFIRMATION
...```

**Rearranging the modules**

APPLICATION

Level 3
execute_session

Level 2
initial transition execute_state is_final

Level 1
display read correct message process
Describing a complete application

No "main program" but class representing a system.

Describe application by remaining features at levels 1 and 2:

- Function transition.
- State initial.
- Boolean function is_final.
- Procedure execute_session.

Describing an application

class APPLICATION
create
make
feature
  initial: INTEGER
  make (n, m: INTEGER) is
  -- Allocate with n states and m possible choices.
  do
    create transition make (1, n, 1, m)
    create states make (1, n)
  end
feature (NONE) -- Representation of transition diagram
  transition ARRAY [STATE]
  -- State transitions
  states ARRAY [STATE]
  -- State for each index

Implementation decisions

- Represent transition by an array transition: n rows (number of states), m columns (number of choices), given at creation
- States numbered from 1 to n: array states yields the state associated with each index
  (Reverse not needed: why?)
- No deferred boolean function is_final, but convention: a transition to state 0 denotes termination.
- No such convention for initial state (too constraining).
  Attribute initial_number.

Array of states: A polymorphic container

states: ARRAY [STATE]

Notations for accessing array element,
  i.e. states[?] in Pascal:
  states.item(?)
  states@i

(Soon in Eiffel: just states[?])
The array of states

(ENQUIRY_ON_FLIGHTS)

(ENQUIRY_ON_SEATS)

(INITIAL)

(CONFIRMATION)

(RESERVATION)

States

Class structure

STATE

INITIAL

RESERVATION

CONFIRMATION

... Other features of APPLICATION

put_state(s, STATE, number, INTEGER) is
-- Enter state s with index number.
require 1 <= number
do number := states.upper
end states.put(number, s)

choose_initial(number, INTEGER) is
-- Define state number number as the initial
-- state.
require 1 <= number
number := states.upper
do first_number := number
end
More features of APPLICATION

```plaintext
put_transition(source, target, label INTEGER) is
   -- Add transition labeled label from state
   -- number source to state number target. require
   1 <= source
   source <= states.upper
   0 <= target
   target <= states.upper
   1 <= label
   label <= transition.upper2
   do
       transition.put(source, label, target)
   end

invariant
   0 <= st_number
   st_number <= n
   transition.upper1 = states.upper
end
```

Open architecture

During system evolution you may at any time:
- Add a new transition (put_transition).
- Add a new state (put_state).
- Delete a state (not shown, but easy to add).
- Change the actions performed in a given state
  - ...

To build an application

Necessary states — instances of STATE — should be available.
Initialize application:
- `create a make(state_count, choice_count)`
- Assign a number to every relevant state `s`:
  - `a put_state(s, n)`
- Choose initial state `n`:
  - `a choose_initial(no)`
Enter transitions:
- `a put_transition(sou, tar, lab)`
May now run:
- `a execute_session`

Note on the architecture

Procedure `execute_session` is not "the function of the system" but just one routine of APPLICATION.

Other uses of an application:
- Build and modify: add or delete state, transition, etc.
- Simulate, e.g. in batch (replaying a previous session's script), or on a line-oriented terminal.
- Collect statistics, a log, a script of an execution.
- Store into a file or data base, and retrieve.

Each such extension only requires incremental addition of routines. Doesn't affect structure of APPLICATION and clients.
The system is open

Key to openness: architecture based on types of the problem's objects (state, transition graph, application).
Basing it on "the" apparent purpose of the system would have closed it for evolution.

Real systems have no top

Object-Oriented Design

It's all about finding the right data abstractions

End of lecture 18