An O-O design example

### A reservation panel

<table>
<thead>
<tr>
<th>Flight sought from</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Santa Barbara</td>
<td>Zurich</td>
</tr>
<tr>
<td>Departure on or after</td>
<td>23 June</td>
</tr>
<tr>
<td>Preferred airline(s):</td>
<td></td>
</tr>
<tr>
<td>Special requirements:</td>
<td></td>
</tr>
</tbody>
</table>

**AVAILABLE FLIGHTS:**

- **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 transition diagram

**Initial**

- **Enquiry on flights**
  - **Enquiry on seats**
  - **Confirmation**
  - **Reservation**
  - **Help**

### A first attempt

```
procedure enquiry_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
  end

process answer

inspect C when

1 then
  goto Exit

2 then
  goto PHelp

3 then
  goto PReservation

... (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

```
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>States</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 (Initial)</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 (Help)</td>
<td>Exit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 (Conf.)</td>
<td>Exit</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 (Reserv.)</td>
<td>Exit</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 (Seats)</td>
<td>Exit</td>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 (flights)</td>
<td>Exit</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
<td></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
  local current_state, choice: INTEGER
  do
    current_state := initial
    repeat
      choice := execute_state (current_state)
      current_state := transition (current_state, choice)
    until is_final (current_state)
  end
end
```

To describe an application

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

Actions in a state

```plaintext
execute_state (current_state, choice: INTEGER) is
  local answer: ANSWER
  do
    repeat
      display (current_state)
      [answer, choice] := read (current_state)
      good := correct (current_state, answer)
      if not good then
        message (current_state, answer)
        if not good then
          message (current_state, answer)
        end
        end
      until good
    end
  process (current_state, answer)
  return choice
end
```
Specification of the remaining routines

- \( \text{display}(s) \) 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.
- \( \text{correct}(s, a) \) returns true if and only if \( a \) is a correct answer for the question asked in state \( s \).
- If so, \( \text{process}(s, a) \) processes answer \( a \).
- If not, \( \text{message}(s, a) \) 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:
- \( \text{execute}_\text{state}(\text{state INTEGER}) : \text{INTEGER} \)
- \( \text{display}(\text{state INTEGER}) \)
- \( \text{read}(\text{state INTEGER}) : [\text{ANSWER, INTEGER}] \)
- \( \text{correct}(\text{state INTEGER}, \text{a ANSWER}) : \text{BOOLEAN} \)
- \( \text{message}(\text{state INTEGER}, \text{a ANSWER}) \)
- \( \text{process}(\text{state INTEGER}, \text{a ANSWER}) \)
- \( \text{is}_{-}\text{final}(\text{state INTEGER}) \)

Data transmission

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

```plaintext
display (current_state: INTEGER) is do
    inspect current_state
    when state1 then ...
    when state2 then ...
    when staten then ...
end
```

Consequences:
- Long and complicated routines.
- Must know about one 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 \( \text{current_state} \) is passed from \( \text{execute}_\text{session} \) (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 \( \text{execute}_\text{session}, \text{display}, \text{execute}_\text{state}, ... \) as library components, since each must know about all interactive applications that may use it.

The visible architecture
The real story

Level 3
execute_session

Level 2
initial
transition
execute_state
is_final

Level 1
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.

Grouping by data abstractions

Class STATE

defined class STATE
feature
choice INTEGER -- User’s selection for next step
input ANSWER -- User’s answer for this step
display is deferred -- Show screen for this step
read is deferred -- Get user's answer and choice, recording them into input and choice.
correct BOOLEAN is input acceptable?
defined
process is deferred -- Process correct input.
require not correct
end
message is deferred -- Display message for erroneous input.
require correct
end
Class STATE

execute_state is
  local
  do
    good: BOOLEAN
    from
    until
    loop
      display
      read
      good := correct
      if not good then
        message
      end
      end
    end
  end
end

Class structure

To describe a state of an application

Introduce new descendant of STATE:

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

Rearranging the modules

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.

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.
Describing an application

```pascal
class APPLICATION
create
make
feature
  initial: INTEGER
  make (n, m: INTEGER) is
    -- Allocate with n states and m possible choices.
    do
      -- Create transition make (i, n, 1, m)
      create transition.make (1, n, 1, m)
      create states.make (1, n)
    end
end
feature (NONE) -- Representation of transition diagram
  transition: ARRAY2[STATE]
    -- State transitions
  states: ARRAY[STATE]
    -- State for each index
```

Array of states: A polymorphic container

```pascal
states: ARRAY[STATE]
Notations for accessing array element,
i.e. states[i] in Pascal:
  -- states.item (i)
  states @ i
(Soon in Eiffel: just states[i])
```

The array of states

```pascal
execute_session
  -- Run one session of application
  execute_session is
  local
    current_state: STATE -- Polymorphic!
    index: INTEGER
  do
    from index := initial
    invariant 0 <= index
    index <= n
    until index = 0
    loop
      current_state := states @ index
      current_state.execute_state
      check
        1 <= current_state.choice
        current_state.choice <= m
      end
      index := transition.item(index, current_state.choice)
    end
  end
end
```

Executing a session

Class structure

```pascal
put_state (s: STATE; number: INTEGER) is
  -- Enter state s with index number.
  require
    1 <= number
    number <= states.upper
  do
    states.put (number, s)
  end
choose_initial (number: INTEGER) is
  -- Define state number number as the initial state.
  require
    1 <= number
    number <= states.upper
  do
    first_number := number
  end
```

Other features of APPLICATION
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
```

Invariants:

```plaintext
0 <= st_number
st_number <= n
transition.upper1 = states.upper
```

To build an application

Necessary states — instances of *STATE* — should be available.

Initialize application:

```plaintext
create a.make (state_count, choice_count)
```

Assign a number to every relevant state `s`:

```plaintext
a.put_state (s, n)
```

Choose initial state `n0`:

```plaintext
a.choose_initial (n0)
```

Enter transitions:

```plaintext
a.put_transition (source, target, label)
```

May now run:

```plaintext
a.execute_session
```

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
  - ...

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 11