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
FIMAA 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
A first attempt

```plaintext
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
  until no error in answer
  process answer
inspect C
when C0 then
goto Exit
when C1 then
goto PHelp
when Cm-1 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>State</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 (Help)</td>
<td>Exit</td>
<td>Return</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 (Conf.)</td>
<td>Exit</td>
<td></td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 (Reserv.)</td>
<td>Exit</td>
<td></td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 (Seats)</td>
<td>Exit</td>
<td></td>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 (flights)</td>
<td>Exit</td>
<td></td>
<td></td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

The transition diagram

New system architecture

Level 3
- execute_session

Level 2
- initial
- transition
- execute_state
- is_final

Level 1
- display
- read
- correct
- message
- process
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 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, INTEGER, INTEGER) is
  local answer: ANSWER
    good: BOOLEAN
    choice: INTEGER
  do
    repeat
      display (current_state)
      [answer, choice] := read (current_state)
      good := correct (current_state, answer)
    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:

execute_state \((\text{state INTEGER})\) INTEGER
display \((\text{state INTEGER})\)
read \((\text{state INTEGER})\) [\(\text{ANSWER, INTEGER}\)]
correct \((\text{state INTEGER} \& \text{ANSWER})\) BOOLEAN
message \((\text{state INTEGER} \& \text{ANSWER})\)
process \((\text{state INTEGER} \& \text{ANSWER})\)
is_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 state_n 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 `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

Level 1
  display
  read
  correct
  message
  process
```
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

Level 3

Level 2

Level 1

execute_session

initial

transition

execute_state

is_final

STATE
display
read
correct
message
process

Class STATE

defered class STATE

feature

choice: INTEGER

input: ANSWER

display is deferred end

read is deferred end

correct: BOOLEAN

message is deferred end

process is deferred end

require not correct deferred end

require correct deferred end

require not correct deferred end

Class STATE

message is deferred end

process is deferred end

require correct deferred end

require not correct deferred end

require correct deferred end

require not correct deferred end
**Class STATE**

```plaintext
execute_state is
  local
good: BOOLEAN
do
  from until
  loop
good
  display
  read
  good := correct
  if not good then
    message
  end
  end
  process
  choice := input.choice
  end
end
```

**Class structure**

```
* STATE
  INITIAL
  RESERVATION
  CONFIRMATION
```

**To describe a state of an application**

Introduce new descendant of STATE:

```plaintext
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
```
Rearranging the modules

APPLICATION

Level 3

execute_session

Level 2

initial transition execute_state is_final

STATE

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.

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

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: ARRAY2 [STATE]
-- State transitions
states: ARRAY [STATE]
-- State for each index

Array of states: A polymorphic container

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

STATES

(ENQUIRY ON_FLIGHTS)
(ENQUIRY ON_SEATS)
(INITIAL)
(CONIRMATION)
(RESERVATION)
Executing a session

execute_session is
local
   current_state: STATE -- Polymorphic
   index: INTEGER
   from:
   invariant
      index >= initial
   current_state = states @ index
   index >= 0
   check
      1 <= current_state.choice
      current_state.choice <= m
   end
   current_state.execute_state
   index := transition.item (index, current_state.choice)
end

Class structure

* STATE
  INITIAL
  RESERVATION
  CONFIRMATION

Other features of APPLICATION

put_state (s: STATE; number: INTEGER) is
   require
      1 <= number
      number <= states.upper
   do
      states.put (number, s)
   end

choose_initial (number: INTEGER) is
   require
      1 <= number
      number <= states.upper
   do
      first_number := number
   end
More features of \textit{APPLICATION}

\begin{verbatim}
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
\end{verbatim}

To build an application

Necessary states — instances of \textit{STATE} — should be available.
Initialize application:
\begin{verbatim}
create a.make (state_count, choice_count)
\end{verbatim}
Assign a number to every relevant state \( x \):
\begin{verbatim}
a.put_state (x, n)
\end{verbatim}
Choose initial state \( n_0 \):
\begin{verbatim}
a.choose_initial (n0)
\end{verbatim}
Enter transitions:
\begin{verbatim}
a.put_transition (sou, tar, lab)
\end{verbatim}
May now run:
\begin{verbatim}
a.execute_session
\end{verbatim}

Open architecture

During system evolution you may at any time:
\begin{itemize}
    \item Add a new transition (\textit{put_transition}).
    \item Add a new state (\textit{put_state}).
    \item Delete a state (not shown, but easy to add).
    \item Change the actions performed in a given state
    \item ...
\end{itemize}
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 database, 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 19