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

The transition diagram

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>Initial</th>
<th>Help</th>
<th>Conf.</th>
<th>Reserv.</th>
<th>Seats</th>
<th>Flights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

0 (Initial) | Exit  | Return |
1 (Help) | Exit  | Help |
2 (Conf.) | Exit  | Confirmation |
3 (Reserv.) | Exit  | Reservation |
4 (Seats) | Exit  | Exit |
5 (Flights) | Exit  | Exit |

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.

```
procedure execute_session
local
  current_state, choice: INTEGER
begin
  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

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

- `display(s)` outputs the screen associated with state `s`.
- `[a, e] := read(s)` reads into `a` the user's answer to the display screen of state `s`, and into `e` the user's exit choice.
- `correct(s, a)` returns true if and only if `a` is a correct answer for the question asked in state `s`.
- If so, `process(s, a)` processes answer `a`.
- If not, `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(state INTEGER) INTEGER`  
- `display(state INTEGER)`  
- `read(state INTEGER; ANSWER, INTEGER)`  
- `correct(state INTEGER; a: ANSWER) BOOLEAN`  
- `message(state INTEGER; a: ANSWER)`  
- `process(state INTEGER; a: ANSWER)`  
- `is_final(state INTEGER)`

Data transmission

All routines share the state as input argument. They must discriminate on that argument, e.g.:
```
execute_state(state INTEGER) INTEGER
  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 `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
The real story

Level 3

Level 2

Level 1

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 \texttt{STATE} as the basic abstract data type (yielding a class).

Among features of a state:

- The routines of level 1 (deferred in \texttt{STATE})
- \texttt{execute\_state}, as above but without \texttt{current\_state} argument.

Grouping by data abstractions

Class \texttt{STATE}

\begin{verbatim}
deferred class \texttt{STATE}

feature

choice INTEGER -- User's selection for next step
input ANSWER -- User's answer for this step
display is -- Show screen for this step
read is -- Get user's answer and user's choice,
          deferred end
correct BOOLEAN -- Input acceptable?
        deferred end

message is -- Display message for erroneous input.
          require not correct
          deferred end

process is -- Process correct input.
          require correct
          deferred end

end
\end{verbatim}
Class `STATE`

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

- `execute_session`
- `initial`
- `transition`
- `execute_state`
- `is_final`

**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

```plaintext
class APPLICATION
create make

feature
  initial: INTEGER
make (n, m: INTEGER)
  -- 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

```plaintext
states: ARRAY[STATE]

Notations for accessing array element,
i.e. states[1] in Pascal:
  states.item (i)
  states @ i

(Soon in Eiffel: just states[i])
```

The array of states

![Diagram of the array of states with states: INITIAL, RESERVATION, CONFIRMATION, and ENQUIRY_ON_FLIGHTS, ENQUIRY_ON_SEATS, INITIAL marked in different colors.]

Executing a session

```plaintext
execute_session
  -- Run one session of application
local
  current_state: STATE
  index: INTEGER

  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
```

Class structure

![Diagram of the class structure with STATE, INITIAL, RESERVATION, CONFIRMATION marked.]

Other features of APPLICATION

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

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

```haskell
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
```

To build an application

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

Initialize application:

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

Assign a number to every relevant state s:

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

Choose initial state n:

```haskell
a.choose_initial (n)
```

Enter transitions:

```haskell
a.put_transition (sou, tar, lab)
```

May now run:

```haskell
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 19