Software Construction

Bertrand Meyer

University of Zurich
September-December 2017

Lecture 2:
OO Design: The Trailer

A software architecture example
Our first pattern example

Multi-panel interactive systems

Plan of the rest of this lecture:
- Description of the problem: an example
- An unstructured solution
- A top-down, functional solution
- An object-oriented solution yielding a useful design pattern
- Analysis of the solution and its benefits

A reservation panel

Flight sought from: Zurich To: Milan
Depart no earlier than: 18 Feb 2018 No later than: 19 Feb 2018

ERROR: Choose a date in the future

Choose next action:
0 – Exit
1 – Help
2 – Further enquiry
3 – Reserve a seat
A reservation panel

Flight sought from: [Zurich] To: [Milan]
Depart no earlier than: [18 Feb 2018] No later than: [19 Feb 2018]

AVAILABLE FLIGHTS: 2
Flt# AF 425  Dep 8:25  Arr 9:45
Flt# EY 082  Dep 7:40  Arr 9:15

Choose next action:
0 – Exit
1 – Help
2 – Further enquiry
3 – Reserve a seat

The transition diagram

Help 1 Initial 1

Confirmation 2
Reservation 2
Help 1

Flight_query 2
Seat_query 2
Help 1

Help 1
A first attempt

A program block for each state, for example:

\[ P_{\text{Flight_query}}: \]
- display “enquiry on flights” screen
- repeat
  - Read user’s answers and his exit choice C
  - if Error_in_answer then output_message end
- until not Error_in_answer
- end
- process answer
- inspect C
  - when 0 then goto \( P_{\text{Exit}} \)
  - when 1 then goto \( P_{\text{Help}} \)
  - ...
  - when n then goto \( P_{\text{Reservation}} \)
- end

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

Represent the structure of the diagram by a function

\[ transition (i, k) \]

giving the state to go to from state \( i \) for choice \( k \)

This describes the transitions of any particular application

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

<table>
<thead>
<tr>
<th></th>
<th>0 (Initial)</th>
<th>1 (Help)</th>
<th>2 (Confirmation)</th>
<th>3 (Reservation)</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></td>
<td></td>
<td>Exit</td>
<td>Return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 (Confirmation)</td>
<td>Exit</td>
<td></td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 (Reservation)</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>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The transition diagram

New system architecture

execute_session

initial transition execute_state is_final
display read correct message process
display read correct message process
New system architecture

Procedure *execute_session* only defines graph traversal
It knows nothing about particular screens of a given application; it should be the same for all applications

```plaintext
execute_session
  -- Execute full 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

\[ \text{execute\_state} (\text{current\_state} : \text{INTEGER}) : \text{INTEGER} \]

-- Execute actions for current\_state; return user’s exit choice.

local

answer : \text{ANSWER}

good : \text{BOOLEAN}

choice : \text{INTEGER}

do

repeat

\[ \text{display} (\text{current\_state}) \]

\[ \text{[answer, choice]} := \text{read} (\text{current\_state}) \]

\[ \text{good} := \text{correct} (\text{current\_state}, \text{answer}) \]

\[ \text{if not good then message} (\text{current\_state}, \text{answer}) \text{ end} \]

until

\[ \text{good} \]

end

\[ \text{process} (\text{current\_state}, \text{answer}) \]

Result := choice

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 (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 it, e.g.:

```plaintext
display (current_state: INTEGER)
do
    inspect current_state
    when state_1 then
        ...
    when state_2 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.

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

The law of inversion

- If your routines exchange too much data, put your routines into your data.

In this example: the state is everywhere!
Going O-O

Use *STATE* as the basic *abstract data type* (and class)

Among features of every state:

- The routines of level 1 (deferred in class *STATE*)

- *execute_state*, as above but without the argument *current_state*
Class **STATE**

defferred class **STATE**

feature

choice : INTEGER
        -- User’s selection for next step
input : ANSWER
        -- User’s answer for this step

display
        -- Show screen for this state.
defered
end

read
        -- Get user’s answer and exit choice,
        -- recording them into input and choice.
defered
ensure
        input /= Void
end

Class **STATE**

correct : BOOLEAN
        -- Is input acceptable?
defered
end

message  
        -- Display message for erroneous input.
require not correct
defered
end

process
        -- Process correct input.
require correct
defered
end
Class **STATE**

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

Class structure
To describe a state of an application

Write a descendant of `STATE`:

```plaintext
class FLIGHT_QUERY inherit STATE
feature
display do ... end
read do ... end
correct : BOOLEAN do ... end
message do ... end
process do ... end
end
```

Rearranging the modules

```
APPLICATION

Level 3
execute_session

Level 2
initial transition execute_state is_final

Level 1
display read correct message process

STATE
```
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)
-- 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

The array of states

A polymorphic data structure!
Executing a session

```
execute_session
   -- Run one session of application.
   local
      current_state : STATE   -- Polymorphic!
      index : INTEGER
   do
      from index := initial
      until index = 0
      loop
         current_state := states [index]
         current_state.execute_state
      end
      index := transition [index, current_state.choice]
   end
```

Class structure

```
STATE
   + execute_state
   + display
   + read
   + correct
   + message
   + process

INITIAL
   + display
   + read
   + correct
   + message
   + process

FLIGHT_QUERY
   + display
   + read
   + correct
   + message
   + process

RESERVATION
   + display
   + read
   + correct
   + message
   + process

...```

Other features of **APPLICATION**

**put_state** \( (s: STATE; number: INTEGER) \)

-- Enter state \( s \) with index \( number \).

**require**

1 <= number

**do**

\( states[number] := s \)

**end**

**choose_initial** \( (number: INTEGER) \)

-- Define state number \( number \) as the initial state.

**require**

1 <= number

**do**

\( first\_number := number \)

**end**

---

More features of **APPLICATION**

**put_transition** \( (source, target, label: INTEGER) \)

-- 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:

\[
\text{create } a.\text{make}(\text{state\_count, choice\_count})
\]

Assign a number to every relevant state $s$:

\[
a [n] := s
\]

Choose initial state $n0$:

\[
a.\text{choose\_initial}(n0)
\]

Enter transitions:

\[
a.\text{put\_transition}(\text{sou, tar, lab})
\]

May now run:

\[
a.\text{execute\_session}
\]

Open architecture

During system evolution you may at any time:

- Add a new transition ($\text{put\_transition}$)
- Add a new state ($\text{put\_state}$)
- 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
The design pattern

“State and Application”

Software architecture: the basic issue

Finding the right data abstractions
What we have seen

A design pattern: State and Application
The role of data abstraction
Techniques for finding good data abstractions