University of Zurich[UZH]

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

3

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

4

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

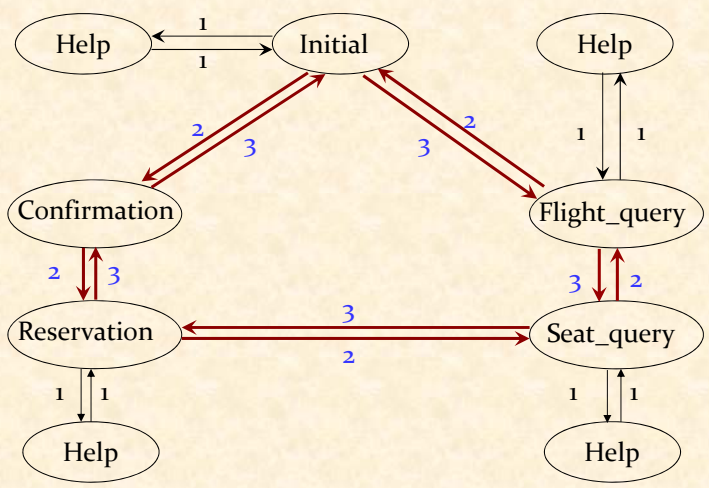
Flt# AF 425	Dep 8:25	Arr 9:45
Flt# EY 082	Dep 7:40	Arr 9:15

Choose next action:

0 – Exit
1 – Help
2 – Further enquiry
3 – Reserve a seat

5

The transition diagram



6

A first attempt

A program block for each state, for example:

```
P_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_Exit
            when 1 then goto P_Help
            ...
            when n then goto P_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

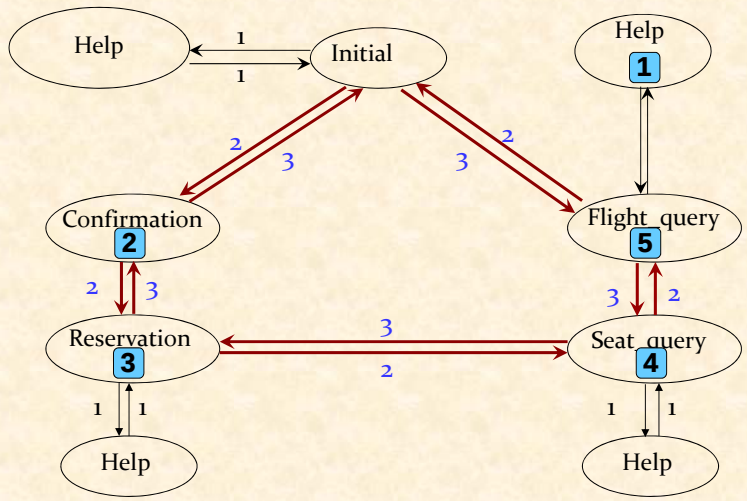
9

The transition function

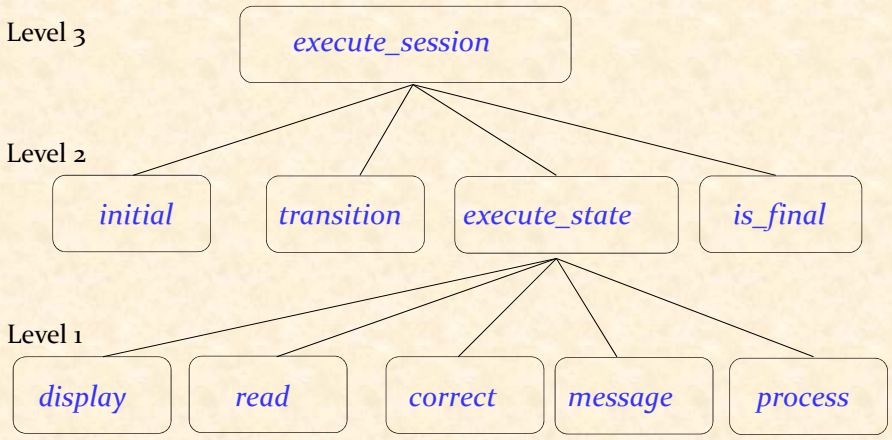
	0	1	2	3
0 (Initial)			2	
1 (Help)	*Exit*	*Return*		
2 (Confirmation)	*Exit*		3	0
3 (Reservation)	*Exit*		4	2
4 (Seats)	*Exit*		5	3
5 (Flights)	*Exit*		0	4

10

The transition diagram



New system architecture



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

```
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
```

13

To describe an application

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

14

Actions in a state

```
execute_state (current_state : INTEGER ): INTEGER
        -- Execute actions for current_state ; return 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 )
                if not good then message (current_state, answer ) end
        until
                good
        end
        process (current_state, answer )
        Result := 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)
```

17

Data transmission

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

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

18

The flow of control

Underlying reason why structure is so inflexible:

Too much DATA TRANSMISSION.

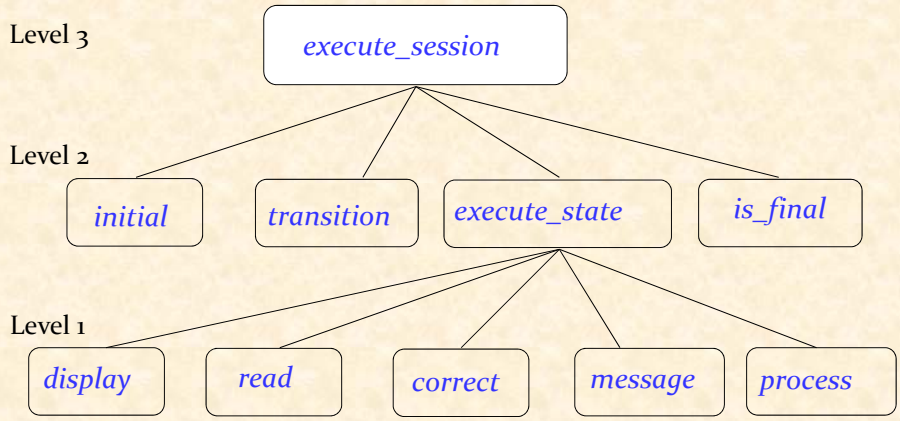
current_state is passed from *execute_session* (level 3) to all routines on level 2 and on to level 1

Worse: there's another implicit argument to all routines – application. Can't define

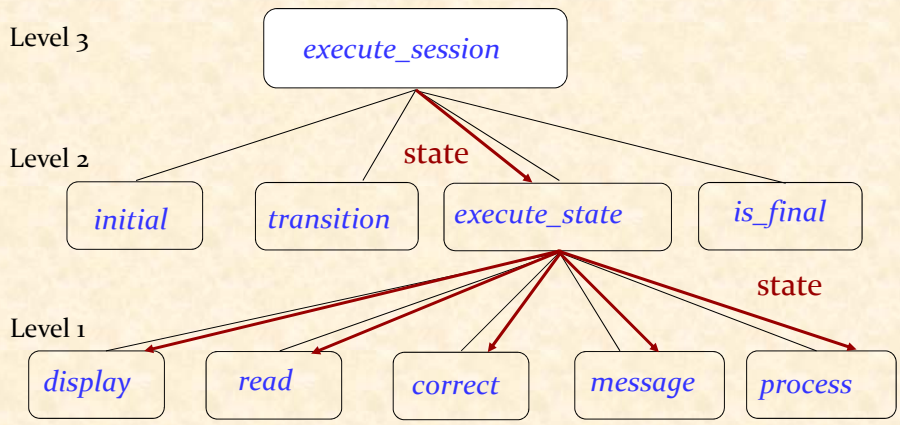
execute_session, *display*, *execute_state*, ...

as library components, since each must know about all interactive applications that may use it.

The visible architecture



The real story



21

The law of inversion

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

In this example: the state is everywhere!

22

Going O-O

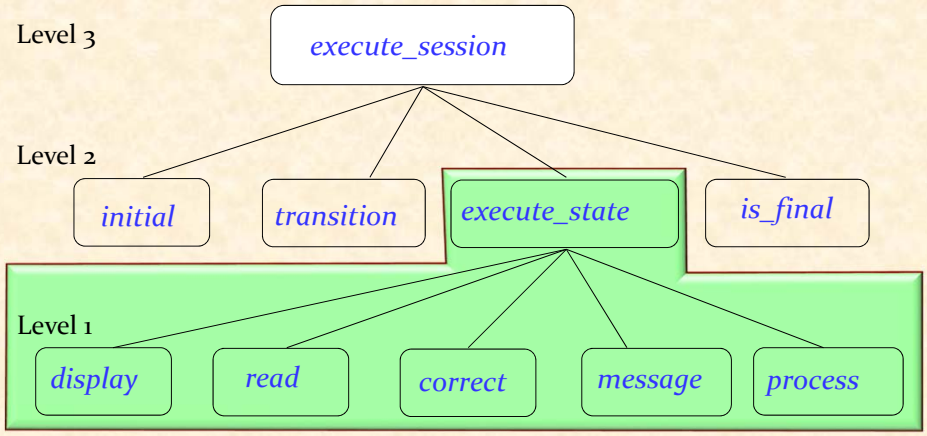
Use *STATE* as the basic abstract data type (and class)

Among features of every state:

- The routines of level 1 (deferred in class *STATE*)
- *execute_state*, as above but without the argument *current_state*

23

Grouping by data abstractions



STATE

24

Class *STATE*

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

25

Class *STATE*

```
correct : BOOLEAN
                -- Is input acceptable?
        deferred
        end

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

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

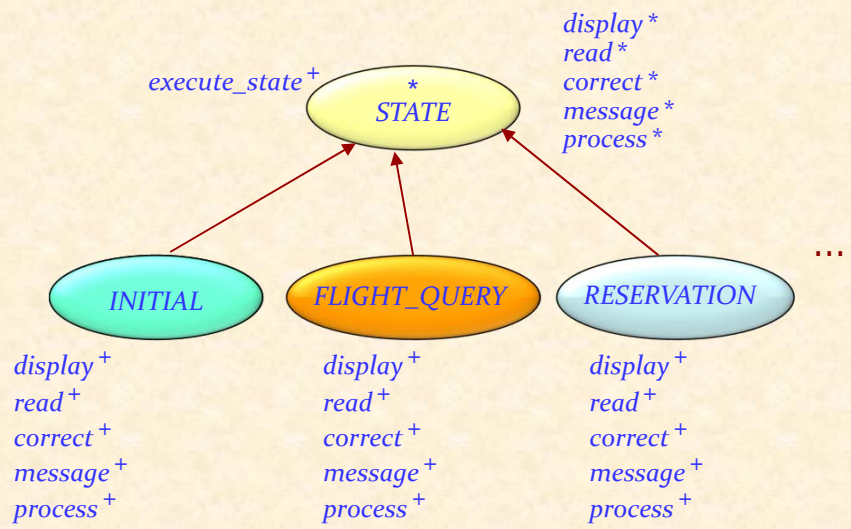
26

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
end
```

27

Class structure



28

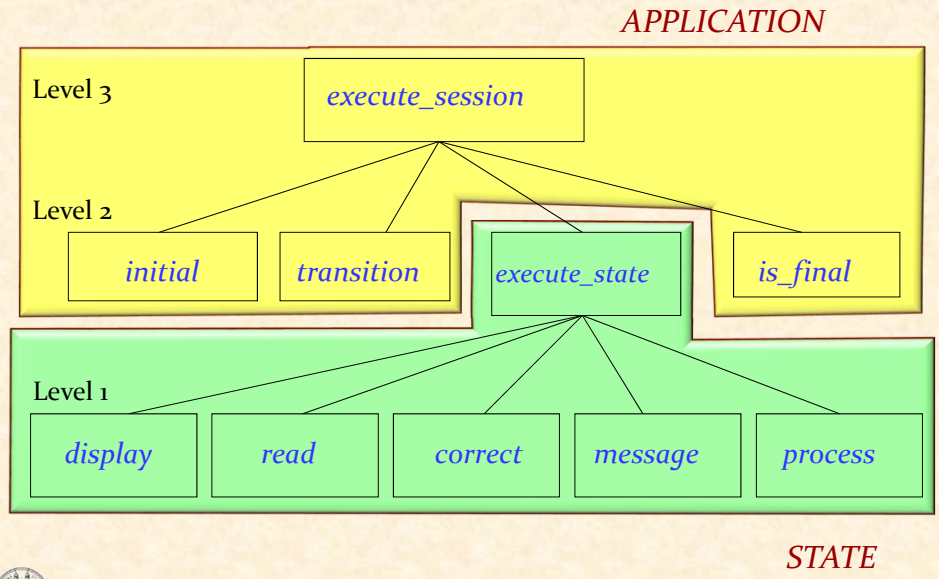
To describe a state of an application

Write a descendant of *STATE* :

```
class FLIGHT_QUERY inherit
        STATE
feature
        display do ... end

        read do ... end

        correct : BOOLEAN do ... end

        message do ... end

        process do ... end
end
```

29

Rearranging the modules



30

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

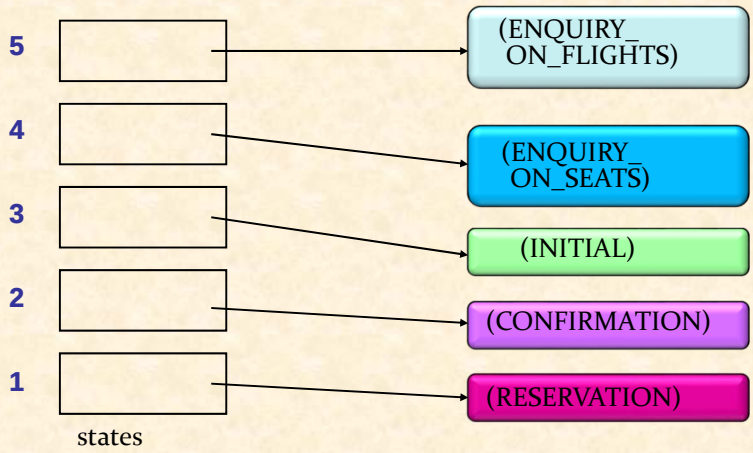
- Represent transition by an array *transition* : *n* rows (number of states), *m* columns (number of choices), given at creation
- States numbered from 1 to *n*; array *states* yields the state associated with each index
- (Reverse not needed: why?)
- No deferred boolean function *is_final*, but convention: 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
```

33

The array of states



A polymorphic data structure!

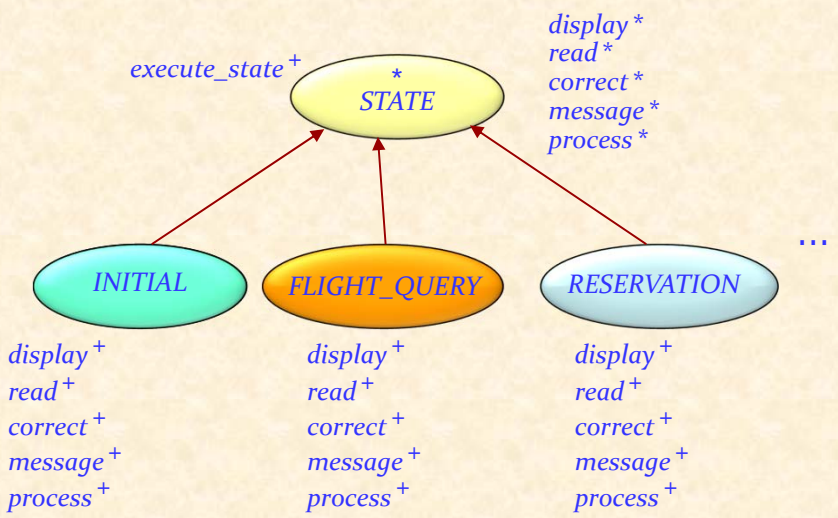
34

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
            index :=transition [index, current_state.choice]
        end
    end
```

35

Class structure



36

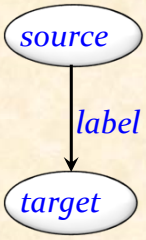
Other features of *APPLICATION*

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

37

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
```



38

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 [*n*] := *s*

Choose initial state *n*0 :

a.choose_initial (*n*0)

Enter transitions:

a.put_transition (*sou*, *tar*, *lab*)

May now run:

a.execute_session

39

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
- …

40

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

41

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

42

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