EECS3311 – Software Design

Polymorphism & the Multi-Panel design

- State Design Pattern
- Template Design Pattern

Multi-panel interactive systems

- Design 1 – hard coded statechart
- Design 2 – top-down structured design
- Design 3 – state pattern

- OOSC2 chapter 20

A reservation panel

-- Enquiry on Flights --

Flight sought from: Toronto To: Zurich

Departure on or after: 23 June On or before: 24 June

Preferred airline (s):

Special requirements:

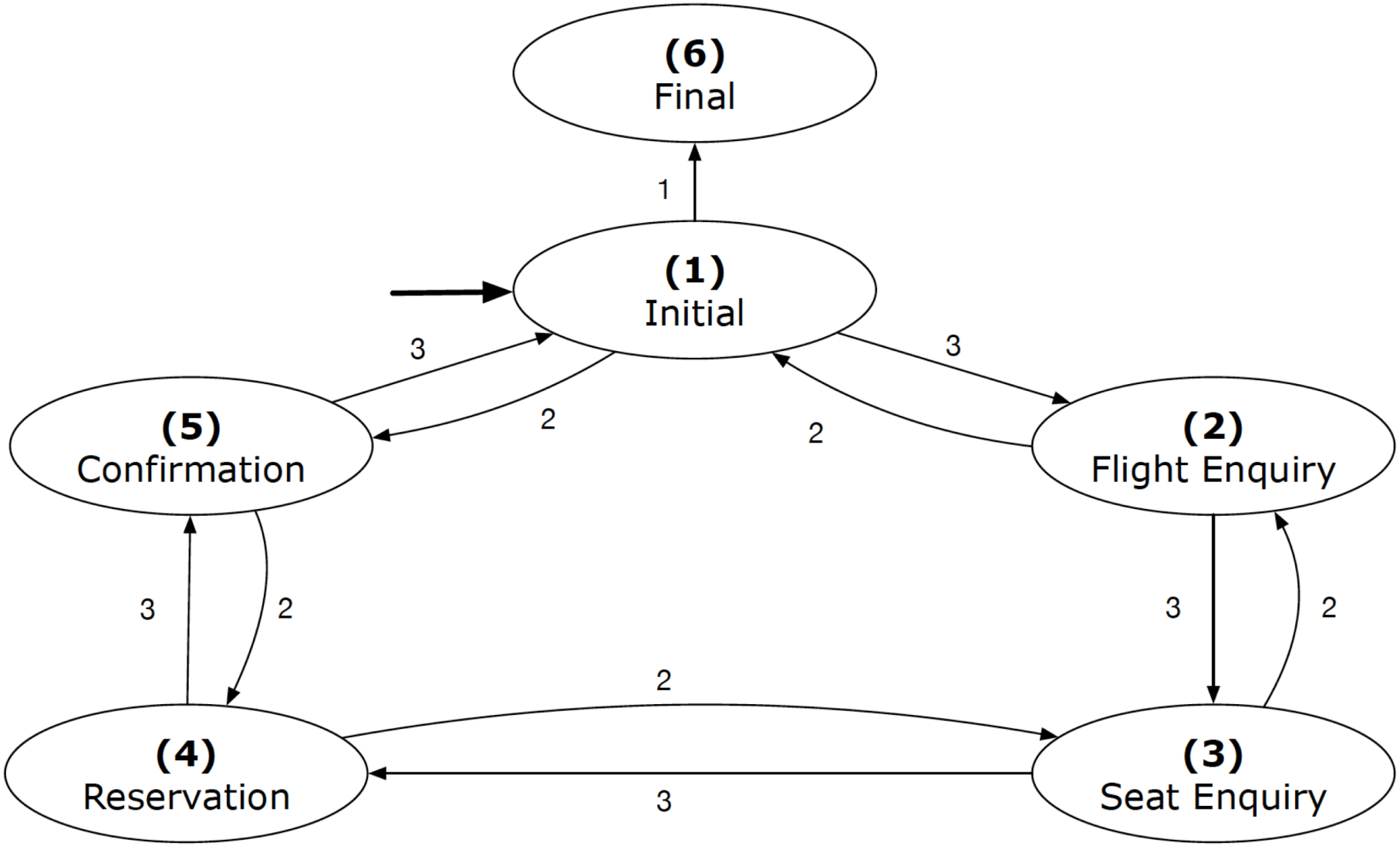
AVAILABLE FLIGHTS: 1

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

Transition diagram (finite state machine)



Design Goals

- [G1] The graph may be large (hundreds of states). Must handle large transition graphs.
- [G2] The structure (e.g. new states and edges) is subject to change. Must be easy to change.
- [G3] Is there a general design pattern so that the same pattern can be re-used for similar systems e.g. an e-commerce system, amazon etc.

A first attempt

```
Seat_Enquiry_Panel:

from
  Display Seat Enquiry Panel
until
  no error in answer
do
  read user's answers
  read user's exit choice C
  if error in answer then
    output message
  end
end
process answer
inspect C
when 2 then
  goto Flight_Enquiry_Panel
when 3 then
   goto Reservation_panel
...
end
```

(and similarly for each state)

```
1_Initial_panel:
  -- Actions for Label 1.
2_Flight_Enquiry_panel:
  -- Actions for Label 2.
3_Seat_Enquiry_panel:
  -- Actions for Label 3.
4_Reservation_panel:
  -- Actions for Label 4.
5_Confirmation_panel:
  -- Actions for Label 5.
6_Final_panel:
  -- Actions for Label 6.
```

What's wrong with the previous scheme?

- Intricate branching "goto" structure ("spaghetti bowl").
- Extendibility problems: dialogue structure hardwired into program structure.
  - Vulnerable to change – new states/transitions will require a change to the system's central control structure

A functional, top-down solution

- Demote/isolate the system's central traversal algorithm into its own generic routine
- Represent the structure of the transition diagram by a function

  transition (*state*, *user_choice:INTEGER*): INTEGER

  -- return new state for **old** state and user_choice

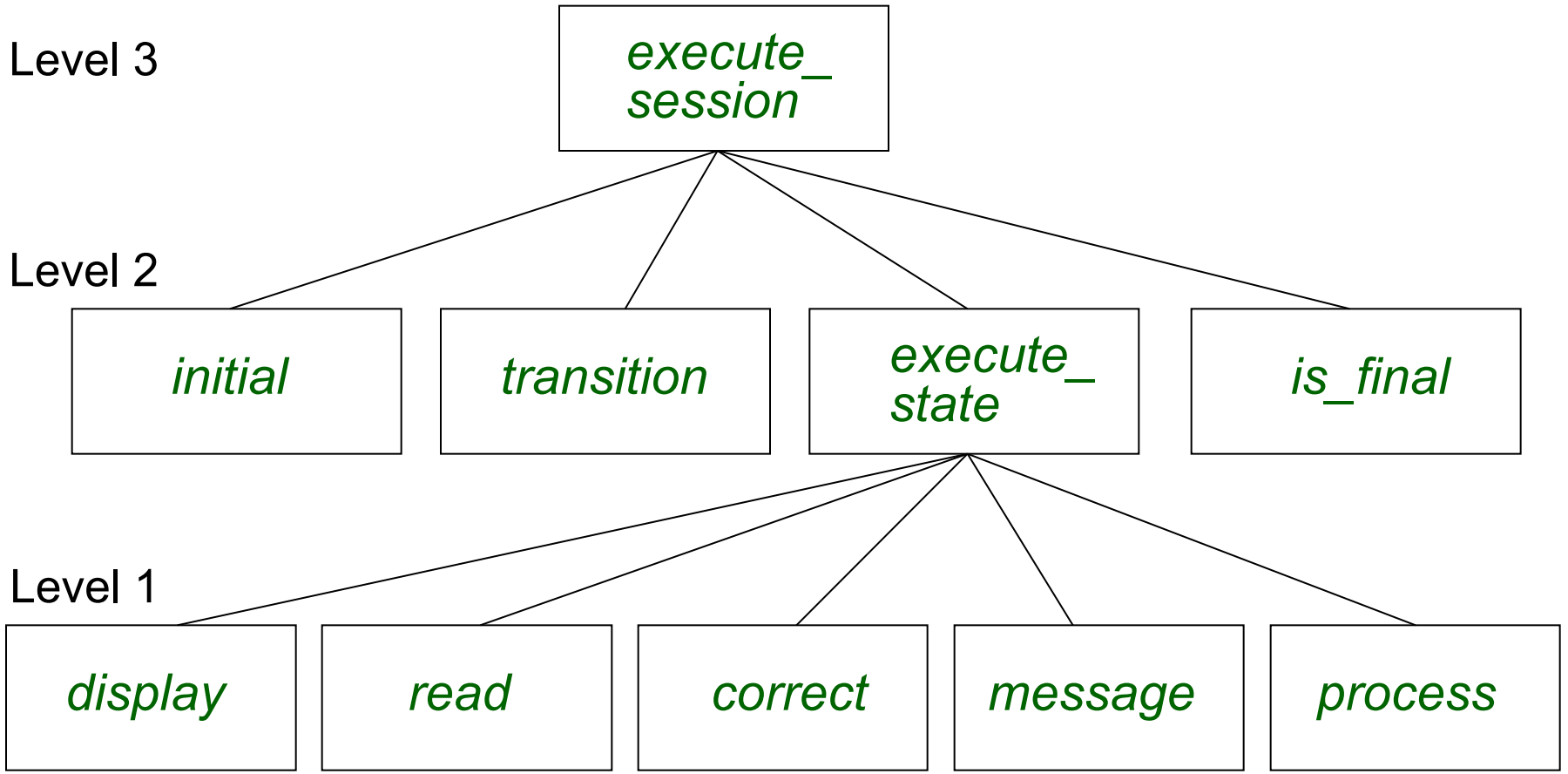
- used to specify the transition diagram associated with any particular interactive application.

- Function *transition* may be implemented as a data structure, for example a two-dimensional array.
  - *transition*: ARRAY2[INTEGER]

transition (*state*, *user_choice:INTEGER*): INTEGER

SRC STATE \ CHOICE	1	2	3
1 (Initial)	6	5	2
2 (Flight Enquiry)	–	1	3
3 (Seat Enquiry)	–	2	4
4 (Reservation)	–	3	5
5 (Confirmation)	–	4	1
6 (Final)	–	–	–

Top Down Decomposition



New system architecture (cont'd)

- Procedure *execute_session* only defines graph traversal.
- Knows nothing about particular screens of a given application. Should be the same for all applications.

```
execute_session
    -- Execute a full interactive session.
  local
    current_state, choice: INTEGER
  do
    from
      current_state := initial
    until
      is_final (current_state)
    do
      choice := execute_state (current_state)
      current_state := transition (current_state, choice)
    end
  end
```

To describe an application

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

Actions in a state

```
execute_state ( current_state : INTEGER) : INTEGER
    -- Handle interaction at the current state.
    -- Return user's exit choice.
  local
    answer: ANSWER; valid_answer: BOOLEAN; choice: INTEGER
  do
    from
    until
      valid_answer
    do
      display( current_state )
      answer := read_answer( current_state )
      choice := read_choice( current_state )
      valid_answer := correct( current_state , answer)
      if not valid_answer then message( current_state , answer)
    end
    process( current_state , answer)
    Result := choice
  end
```

Specification of the remaining routines

FEATURE CALL	FUNCTIONALITY
`display`(**s**)	Display screen outputs associated with **state** *s*
`read_answer`(**s**)	Read user's input for answers associated with **state** *s*
`read_choice`(**s**)	Read user's input for exit choice associated with **state** *s*
`correct`(**s**, answer)	Is the user's *answer* valid w.r.t. **state** *s*?
`process`(**s**, answer)	Given that user's *answer* is valid w.r.t. **state** *s*, process it accordingly.
`message`(**s**, answer)	Given that user's *answer* is not valid w.r.t. **state** *s*, display an error message accordingly.

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/Discriminate on State

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

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

- Consequences:
  - Long/complex routines each discriminating on state
  - Fragile!
  - To change one transition, or add a state, need to change all.

The flow of control

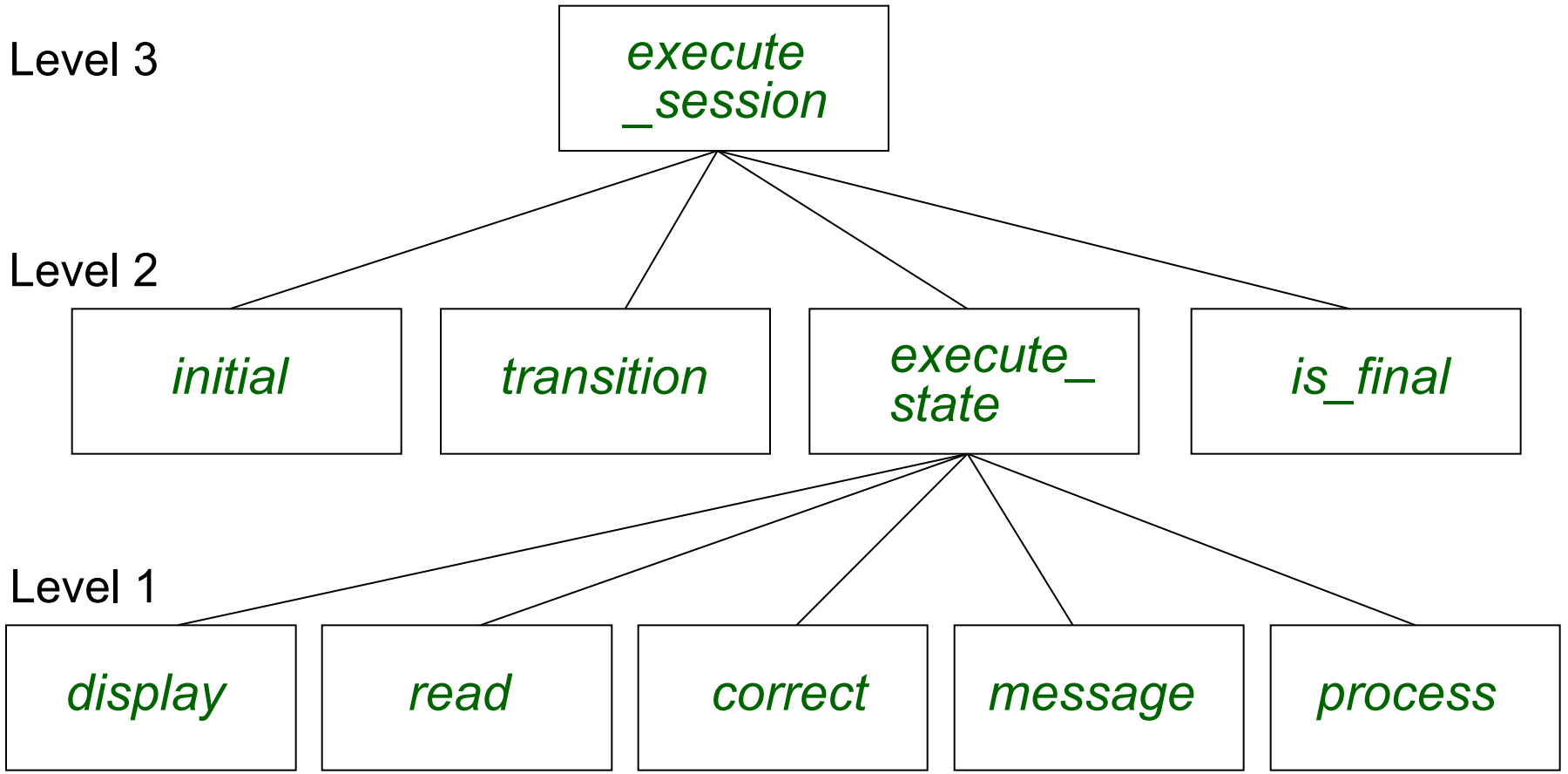
- Underlying reason why structure is so inflexible:

  Too much DATA TRANSMISSION.

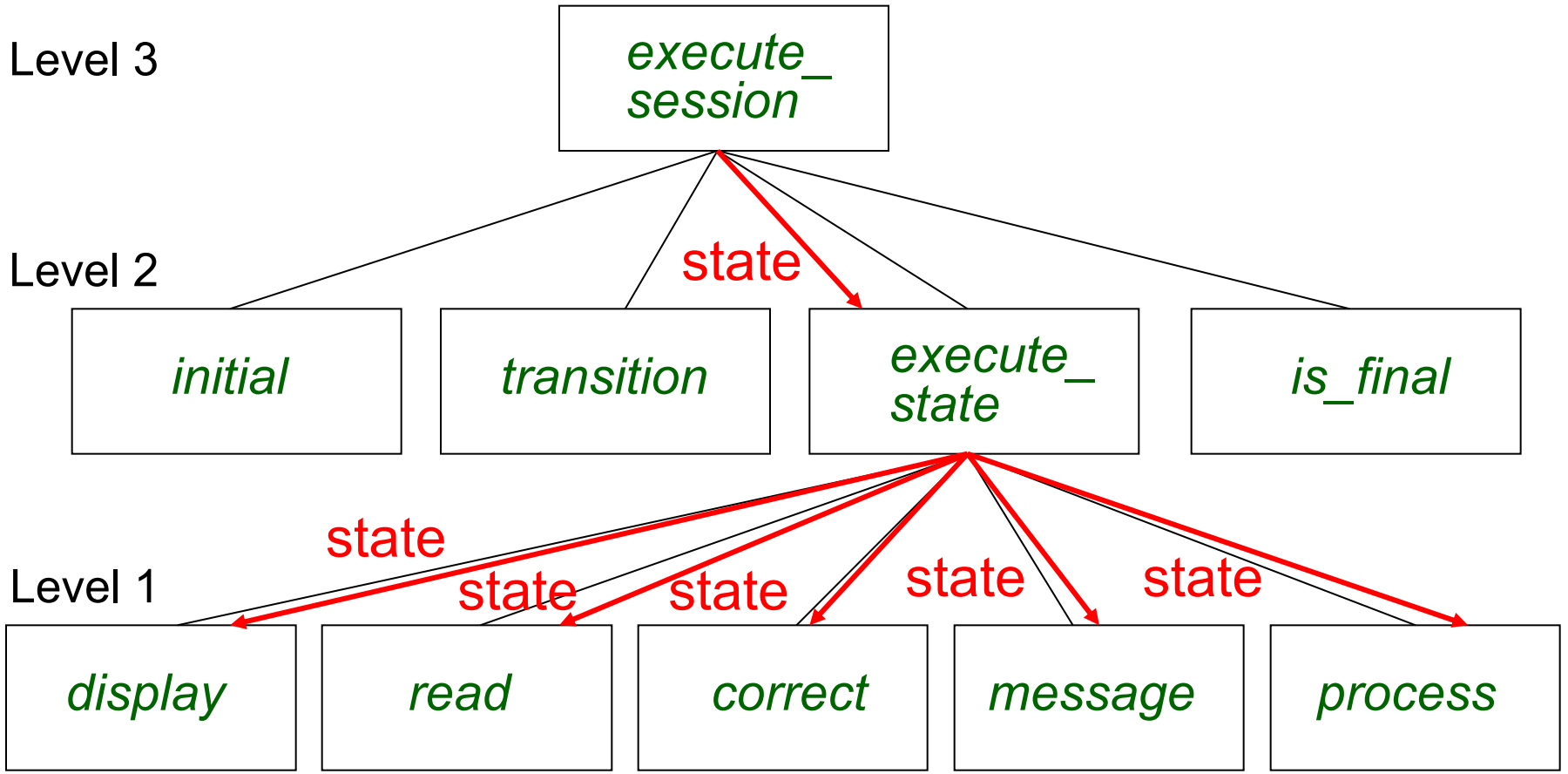
- Variable *current_state* is passed from *execute_session* (level 3) to all routines on level 2 and on to those on level 1.

- 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/State too pervasive!



Going OO

- Architectures/Designs that support
  - Reliability
  - Extendibility
  - Reusability
- What classes?
- What features?
- How do the classes relate to each other?
- How to group the classes into clusters?
- Modularity, Abstraction, Separation of Concerns?
- Information Hiding = Extract what changes and encapsulate it somewhere so that it can change without affecting the rest

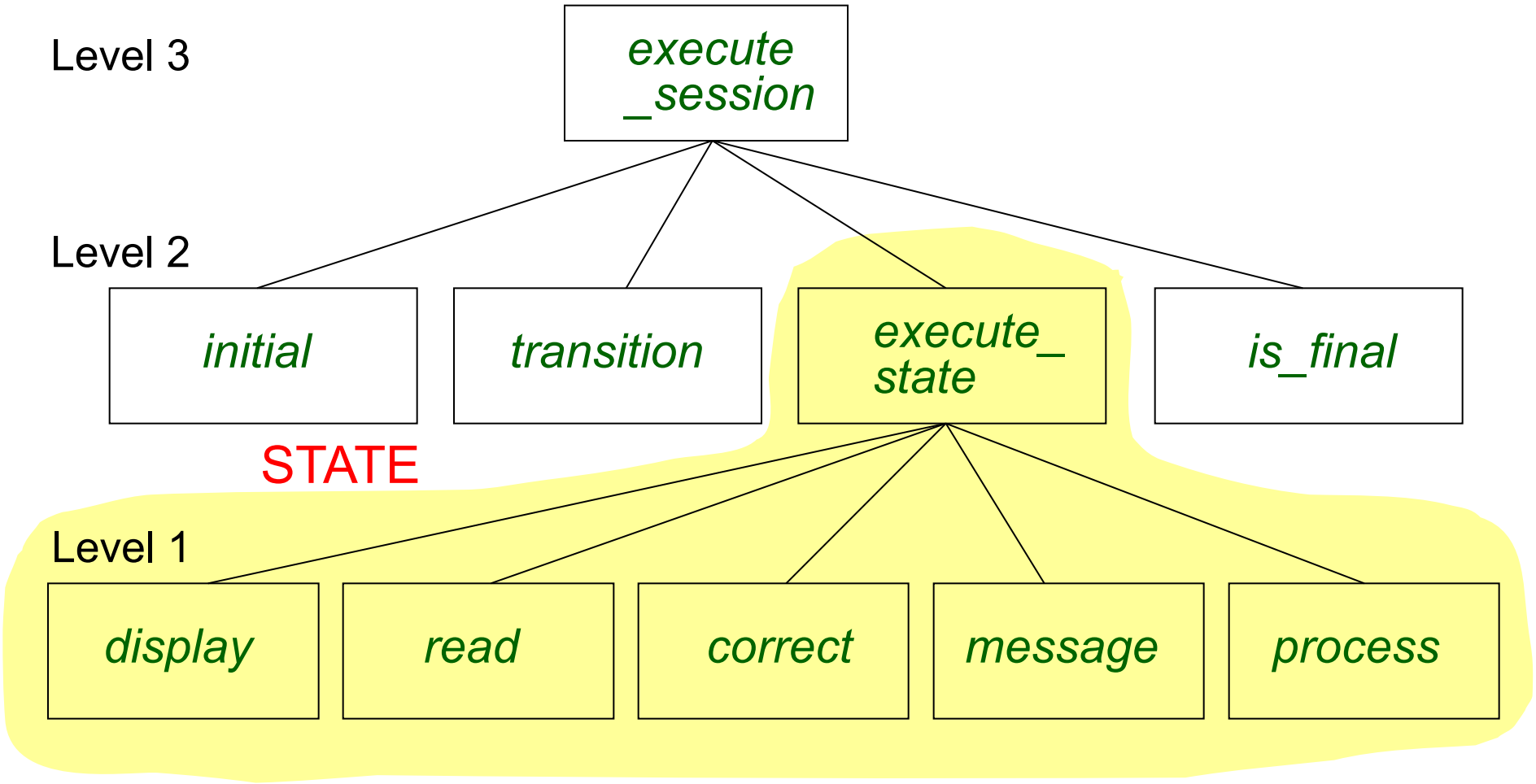
The law of inversion

- The everywhere lurking state
  - If your routines exchange data too much, then 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

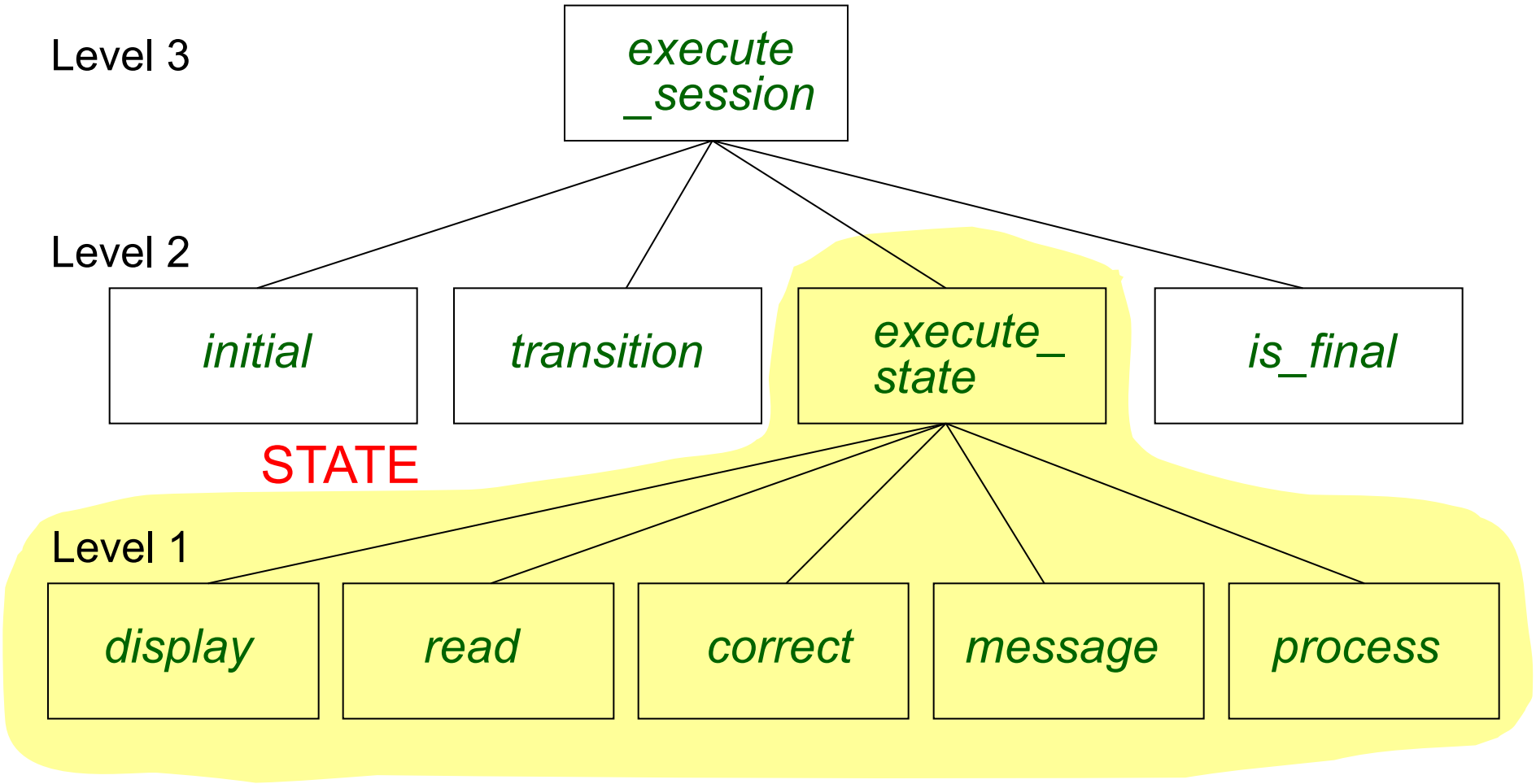


Information Hiding

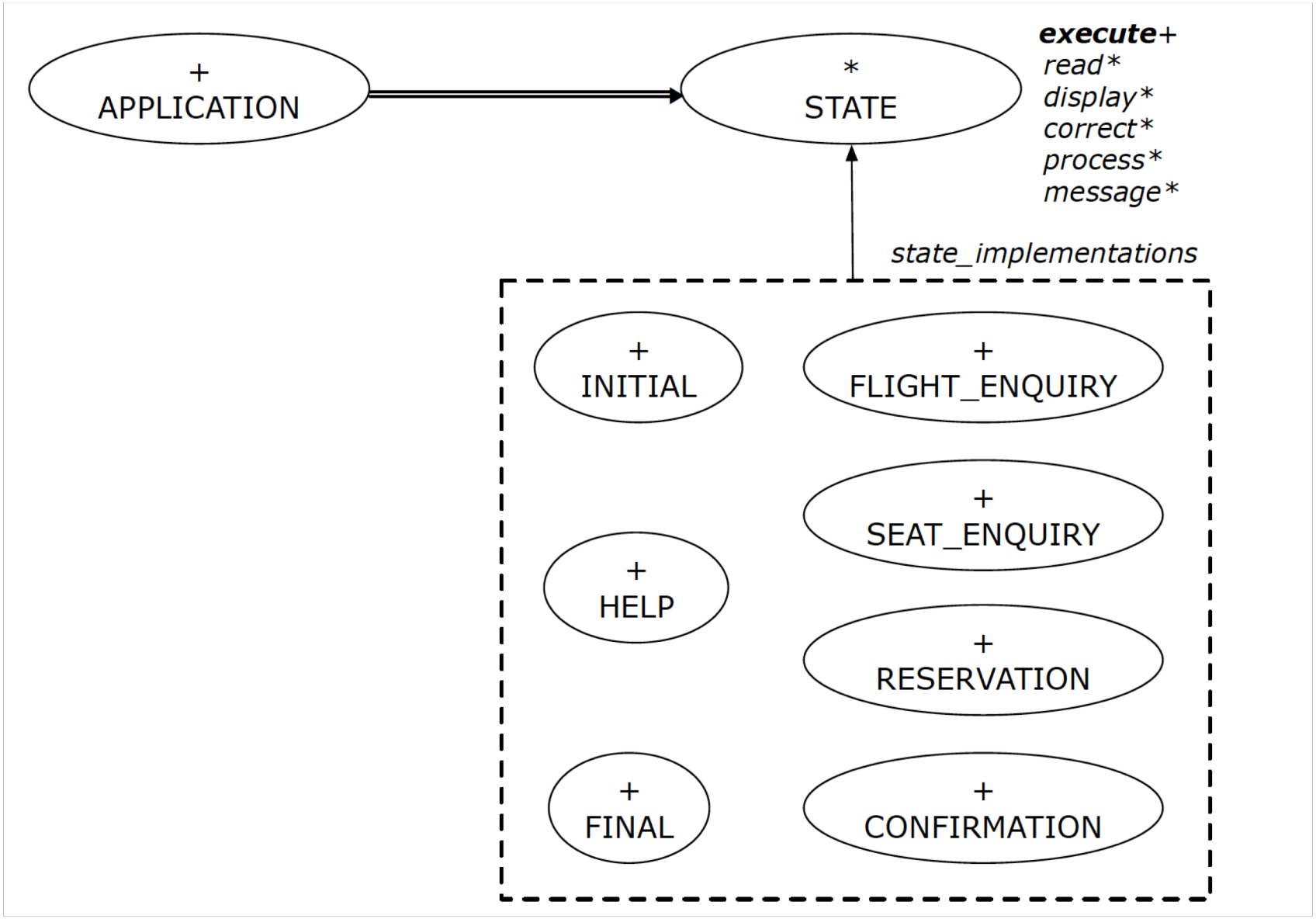
- Take what changes (= state) and encapsulate it (in class STATE) behind a suitable interface (message, process, execute etc.) where it can vary without leaking out and impacting on the rest of the design.

- It was not obvious at first glance that the right abstraction was STATE

Grouping by data abstractions



State Design Pattern



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 step.
            deferred
            end

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

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

Class STATE (cont'd)

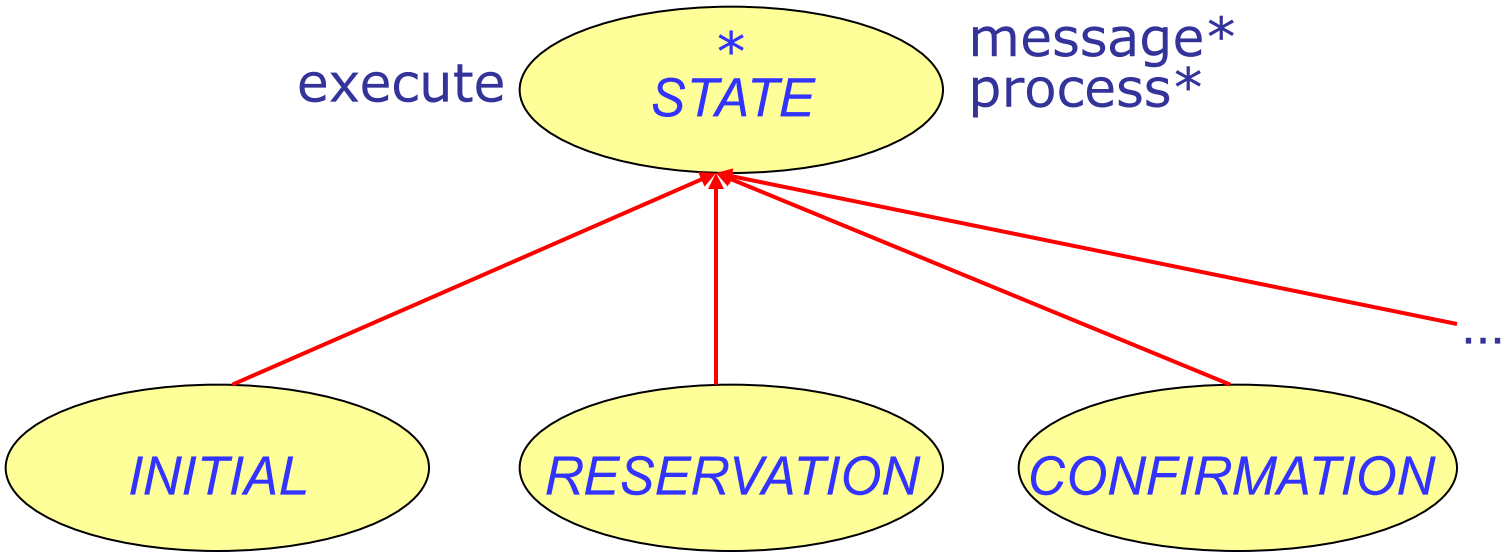
```
message
        -- Display message for erroneous input.
    require
        not correct
    deferred
    end

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

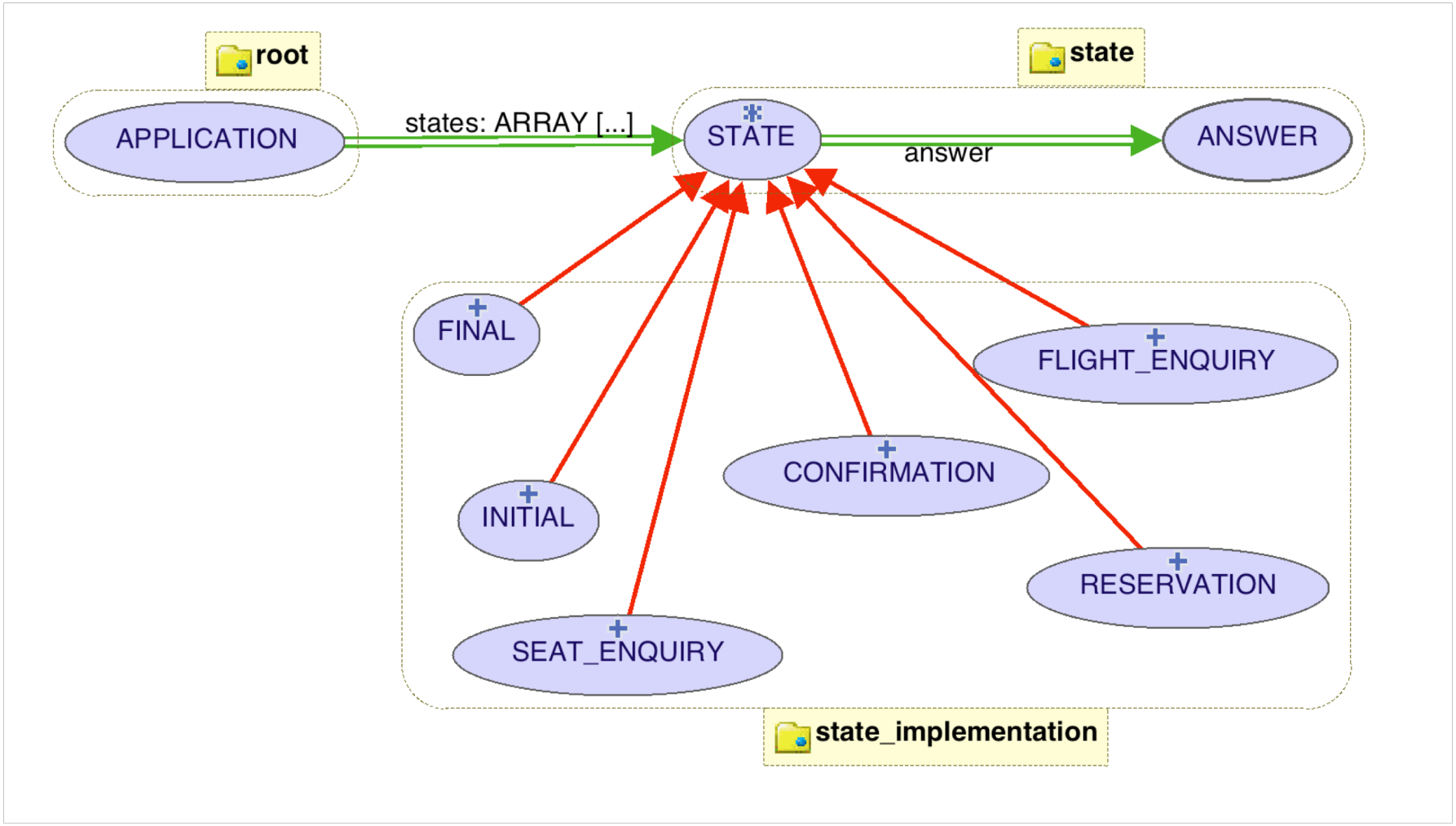
Class STATE (end)

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

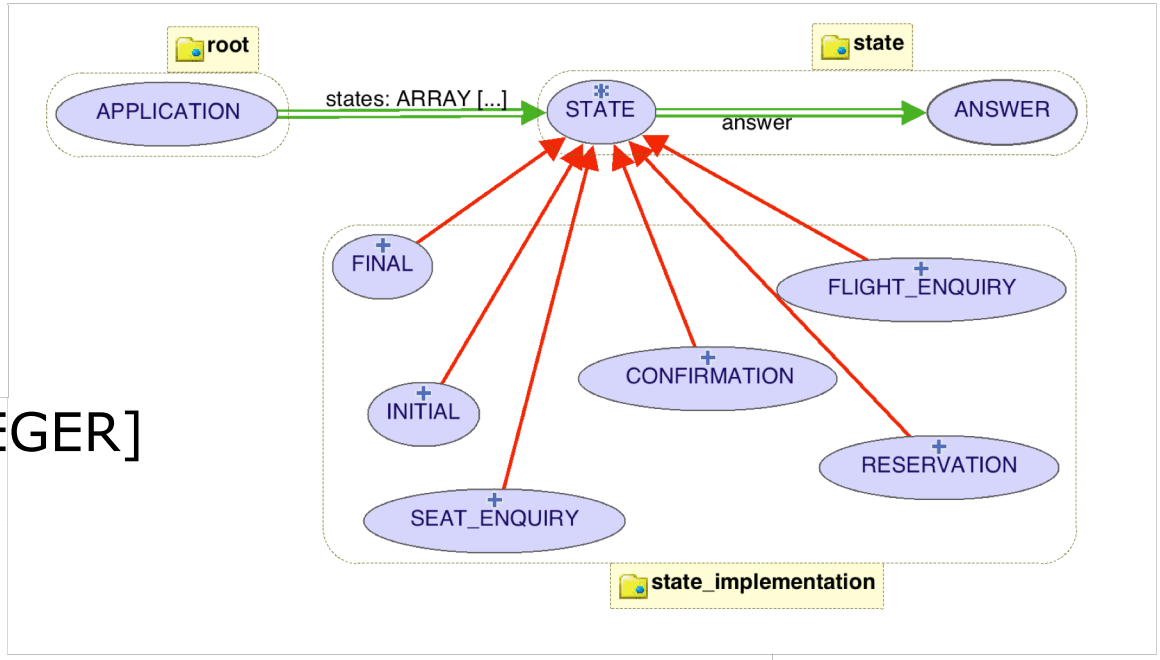
Template Design Pattern: Define the skeleton of an algorithm in a routine, deferring some steps to client subclasses.

Class structure




root
APPLICATION
states: ARRAY [...]
state
STATE
answer
ANSWER
FINAL
FLIGHT_ENQUIRY
CONFIRMATION
INITIAL
RESERVATION
SEAT_ENQUIRY
state_implementation

SRC STATE \ CHOICE	1	2	3
1 (Initial)	6	5	2
2 (Flight Enquiry)	–	1	3
3 (Seat Enquiry)	–	2	4
4 (Reservation)	–	3	5
5 (Confirmation)	–	4	1
6 (Final)	–	–	–



transition: ARRAY2[INTEGER]

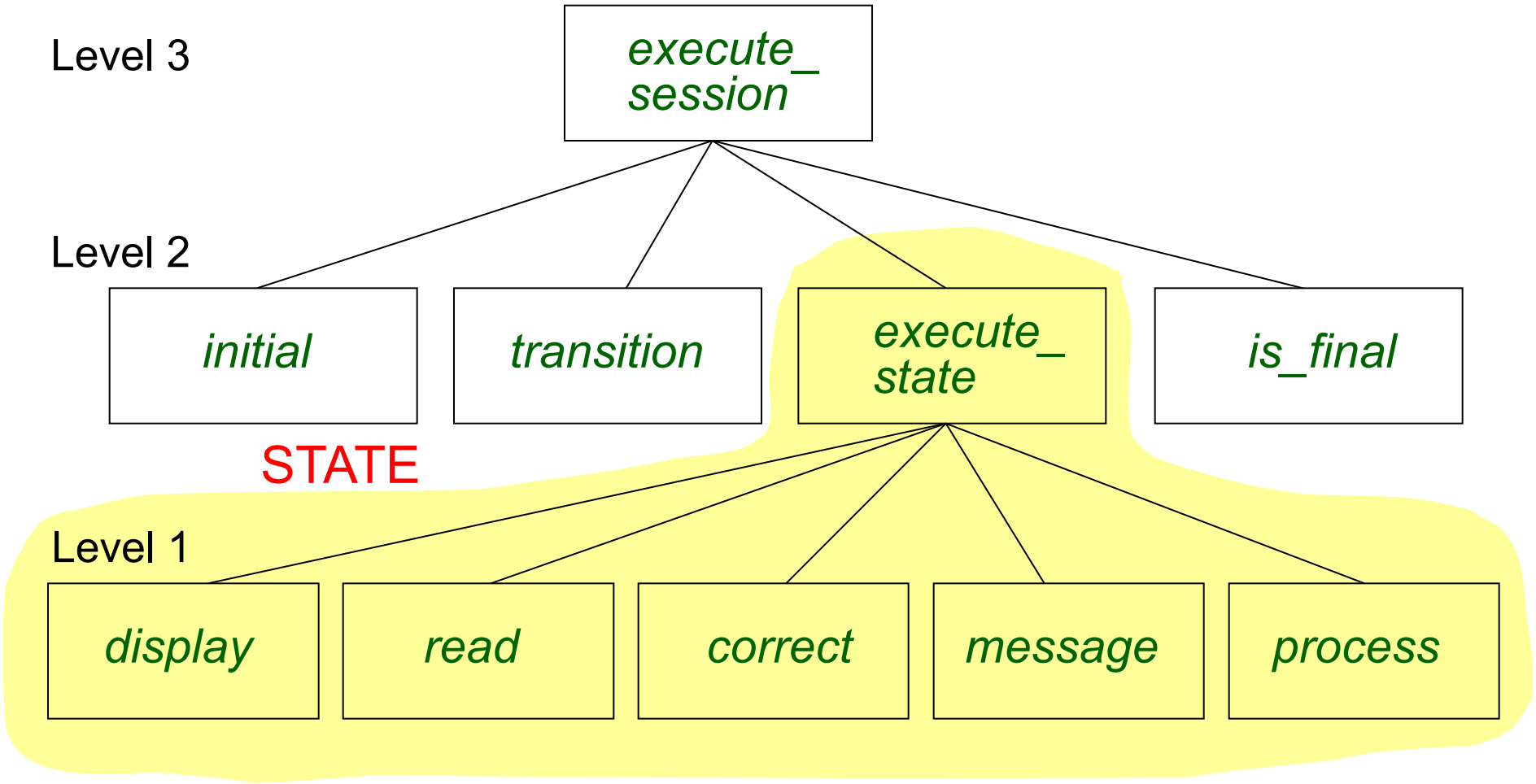
```
class
    APPLICATION
create
    make

feature {TEST_APPLICATION} -- Implementation of Transition Graph
  transition: ARRAY2[INTEGER]
    -- State transitions: transition[state, choice]
  states: ARRAY[STATE]
    -- State for each index, constrained by size of `transition'

feature
  initial: INTEGER
  num_of_states: INTEGER
  num_of_choices: INTEGER

  make(n, m: INTEGER)
    do
        num_of_states := n
        num_of_choices := m
        create transition.make_filled (0, num_of_states, num_of_choices)
        create states.make_empty
    end
```

To describe a state of an application

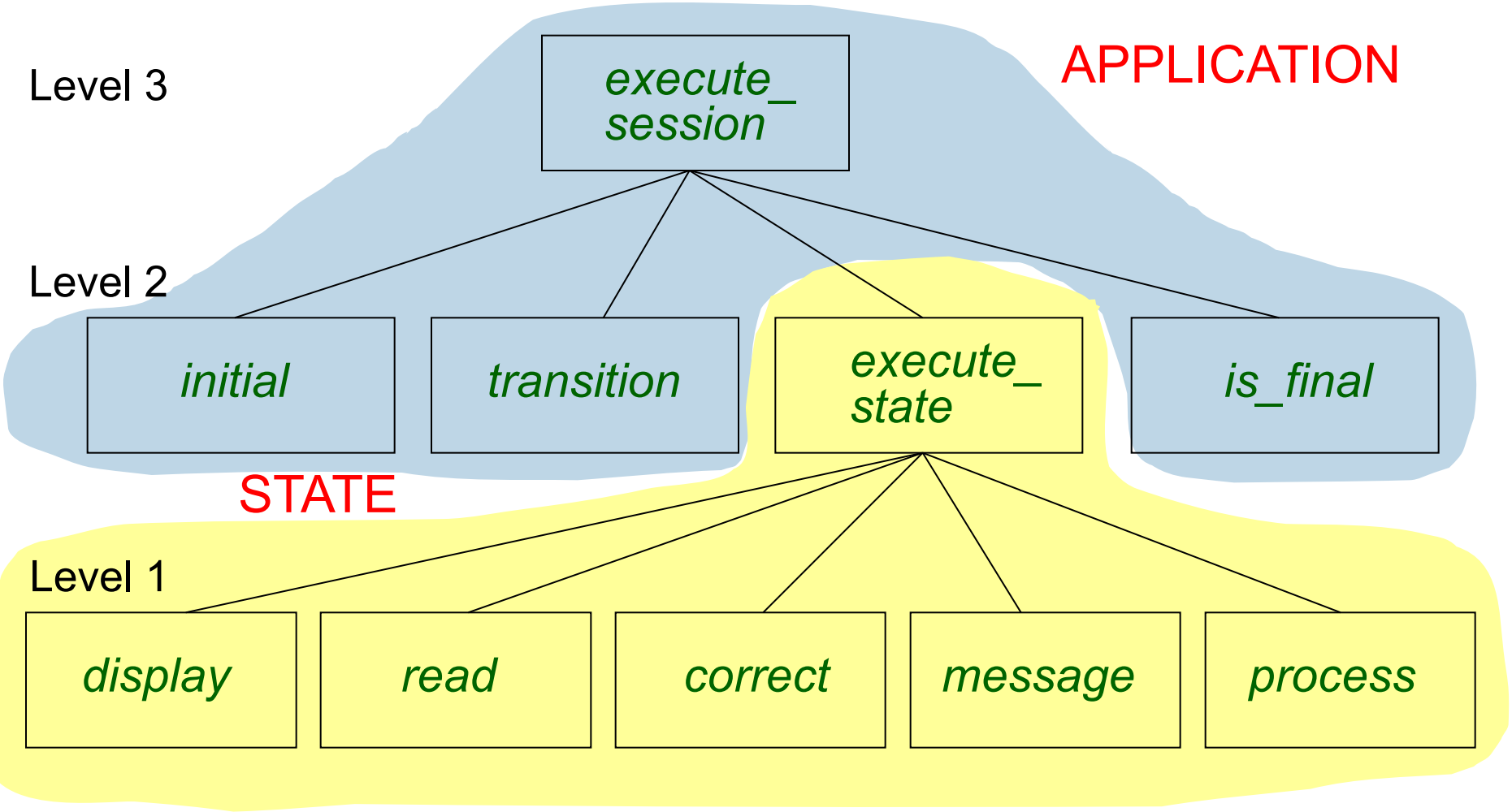
- Introduce new descendant of *STATE*:

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

Rearranging the modules (1)



Rearranging the modules (2)



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*.

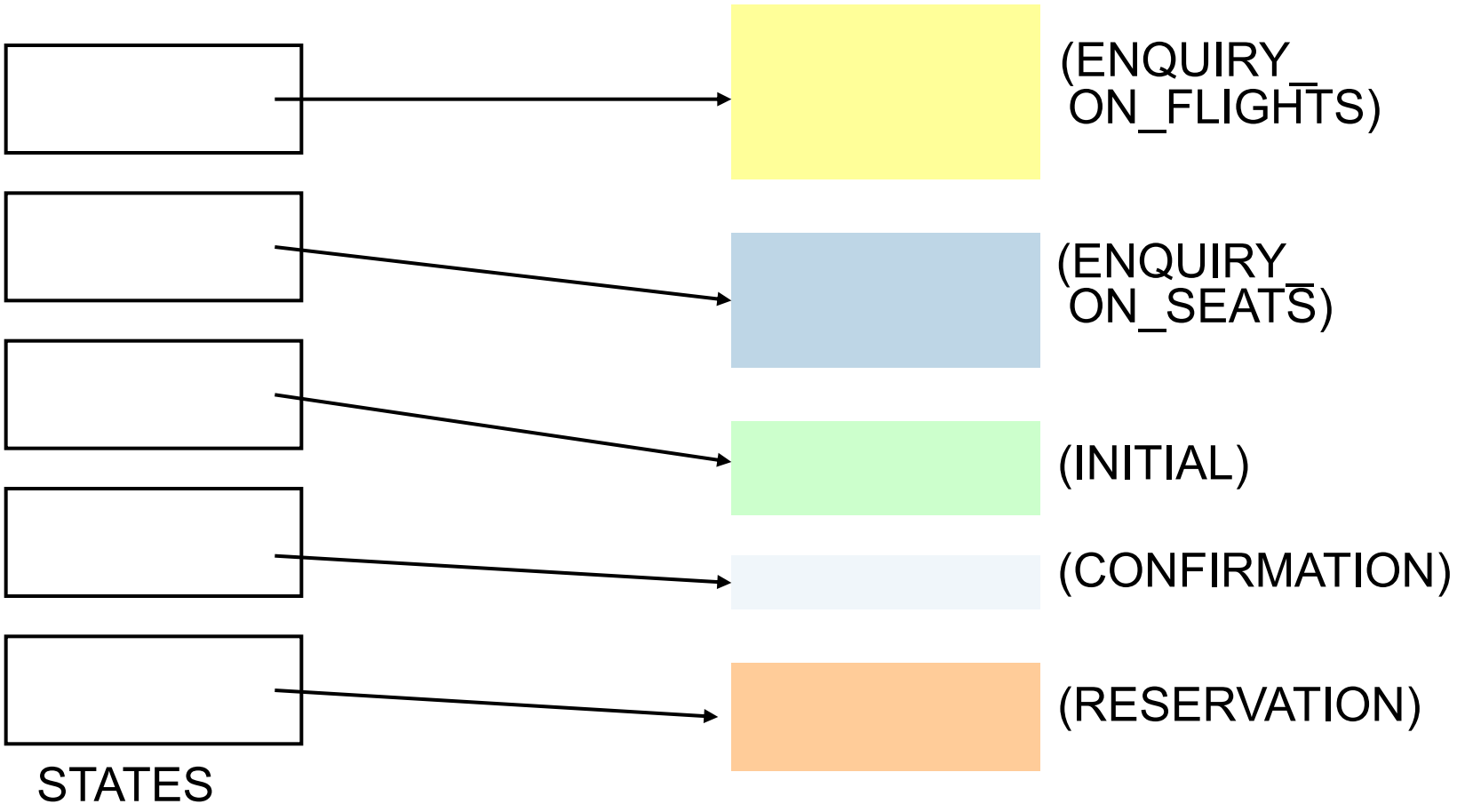
Implementation decisions

- 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*.

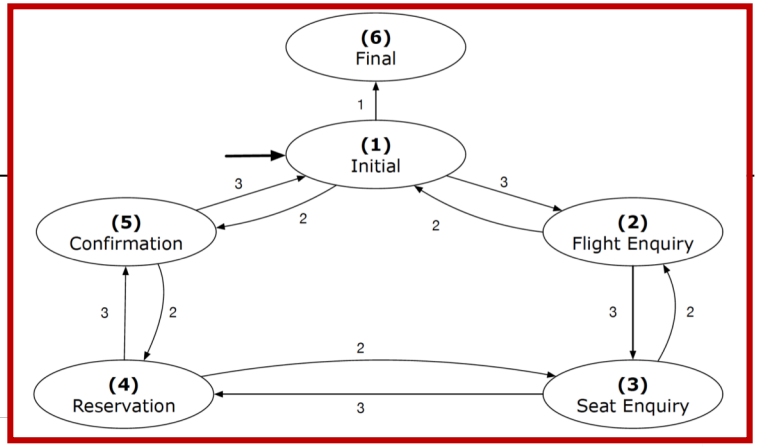
Describing an application (1)

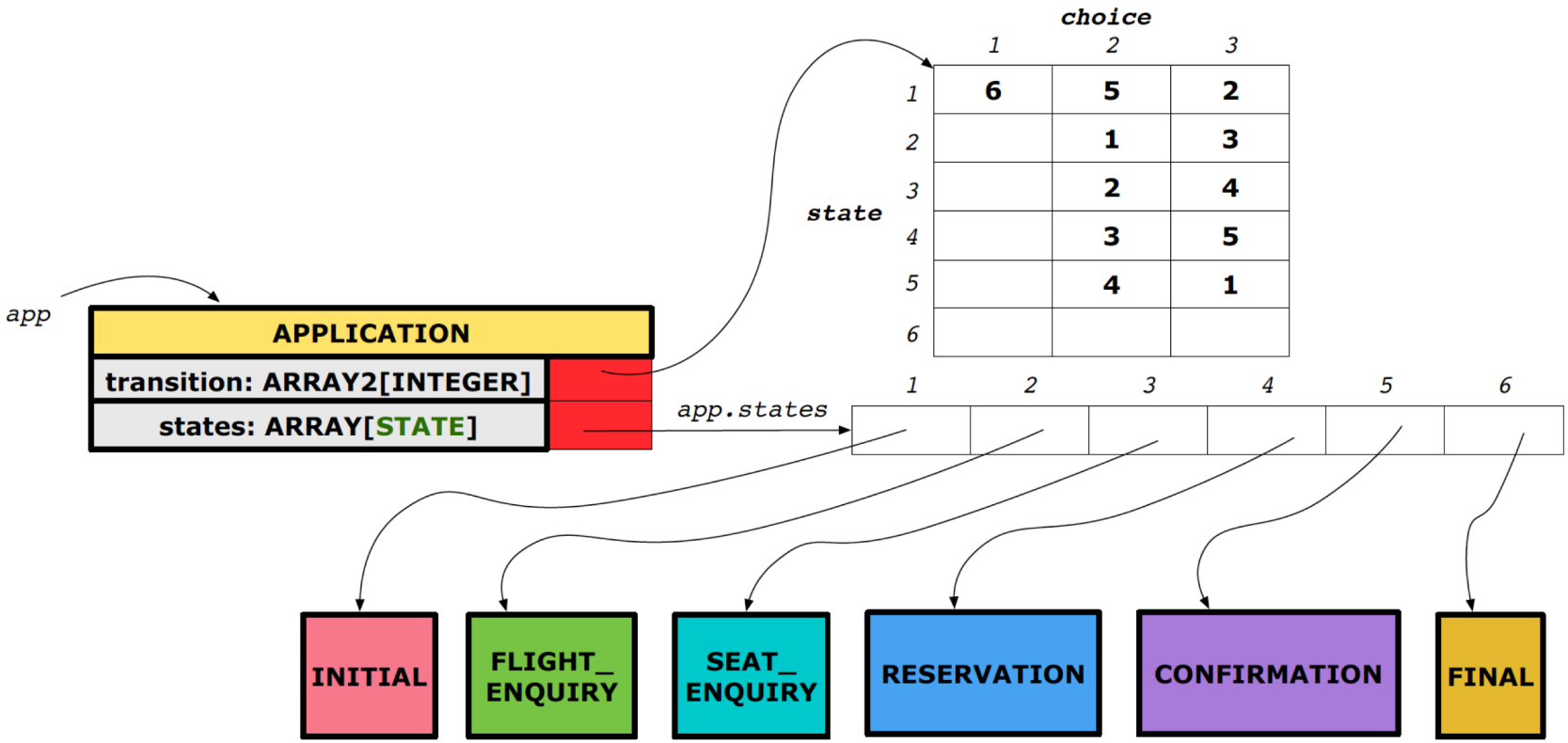
```
class APPLICATION create make
feature {NONE} -- Implementation of Transition Graph
  transition: ARRAY2[INTEGER]
    -- State transitions: transition[state, choice]
  states: ARRAY[STATE]
    -- State for each index, constrained by size of 'transition'
feature
  initial: INTEGER
  number_of_states: INTEGER
  number_of_choices: INTEGER
  make(n, m: INTEGER)
    do number_of_states := n
       number_of_choices := m
       create transition.make_filled(0, n, m)
       create states.make_empty
    end
invariant
  transition.height = number_of_states
  transition.width = number_of_choices
end
```

The array of states



states: ARRAY[STATE]





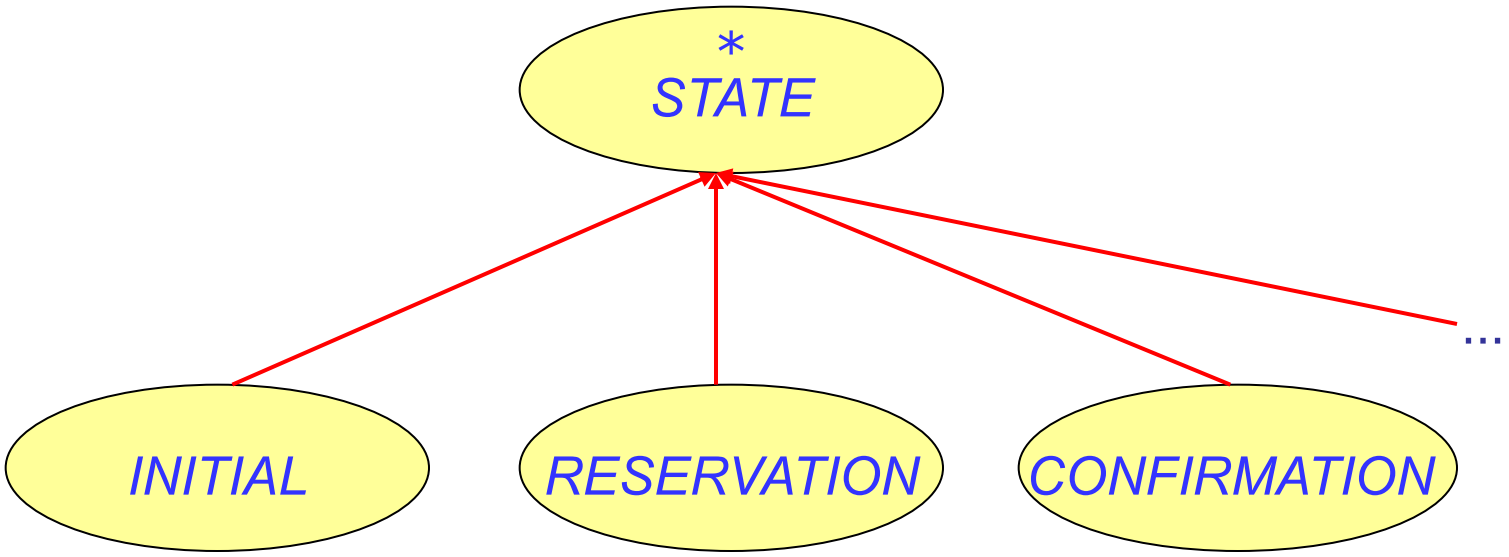
Describing an application (2)

```
class APPLICATION
feature {NONE} -- Implementation of Transition Graph
  transition: ARRAY2[INTEGER]
  states: ARRAY[STATE]
feature
  put_state(s: STATE; index: INTEGER)
    require 1 ≤ index ≤ number_of_states
    do states.force(s, index) end
  choose_initial(index: INTEGER)
    require 1 ≤ index ≤ number_of_states
    do initial := index end
  put_transition(tar, src, choice: INTEGER)
    require
      1 ≤ src ≤ number_of_states
      1 ≤ tar ≤ number_of_states
      1 ≤ choice ≤ number_of_choices
    do
      transition.put(tar, src, choice)
    end
end
```

Describing an application (3)

```
class APPLICATION
feature {NONE} -- Implementation of Transition Graph
  transition: ARRAY2[INTEGER]
  states: ARRAY[STATE]
feature
  execute_session
    local
      current_state: STATE
      index: INTEGER
    do
      from
        index := initial
      until
        is_final (index)
      loop
        current_state := states[index]  -- polymorphism
        current_state.execute  -- dynamic binding
        index := transition.item (index, current_state.choice)
      end
    end
end
```

Class structure



Array of states: A polymorphic container

states: *ARRAY* [*STATE*]

- Notations for accessing array element,
  - *states* [*i*]
  - *states*.*item* (*i*)
  - *states* @ *i*

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.*put_state* (*s*, *n*)
- Choose initial state n_0:

  a.*choose_initial* (n_0)
- Enter transitions:

  a.*put_transition* (*source*, *target*, *choice*)
- May now run:

  a.*execute_session*

```
test_application: BOOLEAN
    local
        app: APPLICATION
        current_state: STATE
        index: INTEGER
    do
        create app.make (6, 3)
        app.put_state (create {INITIAL}.make, 1)
        app.put_state (create {FLIGHT_ENQUIRY}.make, 2)
        app.put_state (create {SEAT_ENQUIRY}.make, 3)
        app.put_state (create {RESERVATION}.make, 4)
        app.put_state (create {CONFIRMATION}.make, 5)
        app.put_state (create {FINAL}.make, 6)

        app.choose_initial (1)

        app.put_transition (6, 1, 1)

        app.put_transition (2, 1, 3)
        app.put_transition (3, 2, 3)
        app.put_transition (4, 3, 3)
        app.put_transition (5, 4, 3)
        app.put_transition (1, 5, 3)

        app.put_transition (5, 1, 2)
        app.put_transition (4, 5, 2)
        app.put_transition (3, 4, 2)
        app.put_transition (2, 3, 2)
        app.put_transition (1, 2, 2)

        index := app.initial
        current_state := app.states [index]
        Result := attached {INITIAL} current_state
        check Result end

        -- Say user's choice is 3: transit from INITIAL to FLIGHT_STATUS
        index := app.transition.item (index, 3)
        current_state := app.states [index]
        Result := attached {FLIGHT_ENQUIRY} current_state
        check Result end
    end
```

Single Choice Principle

Constrained by invariants

An example

```
test_application: BOOLEAN
  local
    app: APPLICATION ; current_state: STATE ; index: INTEGER
  do
    create app.make (6, 3)
    app.put_state (create {INITIAL}.make, 1)
    -- Similarly for other 5 states.
    app.choose_initial (1)
    -- Transit to FINAL given current state INITIAL and choice 1.
    app.put_transition (6, 1, 1)
    -- Similarly for other 10 transitions.

    index := app.initial
    current_state := app.states [index]
    Result := attached {INITIAL} current_state
    check Result end
    -- Say user's choice is 3: transit from INITIAL to FLIGHT_STATUS
    index := app.transition.item (index, 3)
    current_state := app.states [index]
    Result := attached {FLIGHT_ENQUIRY} current_state
end
```

transition:ARRAY2[INTEGER]

```
make
        -- Run application.
        -- (export status {NONE})
    do
        create transition.make_filled (0, 3, 3)
        check
                transition [1, 1] = 0
        end
        transition [1, 1] := 1
        transition [3, 3] := 9
        check
                transition [1, 1] + transition [1, 9] = 10
        end
    end
```

Feature | Class

Objects — {APPLICATION}.make

Name	Value	Type	Address
Current object	<0x10E977A58>	APPLICATION	0x10E977A58
transition	<0x10E977A68>	ARRAY2 [INTEGER_32]	0x10E977A68
area	count=9, capacity=9	SPECIAL [INTEGER_32]	0x10E977A80
count	9		
capacity	9		
0	1	INTEGER_32	
1	0	INTEGER_32	
2	0	INTEGER_32	
3	0	INTEGER_32	
4	0	INTEGER_32	
5	0	INTEGER_32	
6	0	INTEGER_32	
7	0	INTEGER_32	
8	9	INTEGER_32	

ARRAY2

```
transition: ARRAY2 [INTEGER_32]

make
        -- Run application.
    do
        create transition.make_filled (0, 3, 3)
        check
                transition [1, 1] = 0
        end
        transition [1, 1] := 1
        transition [3, 3] := 9
        check
                transition [1, 1] + transition [1, 9] = 10
        end
    end
```

```
item alias "[]" (row, column: INTEGER_32): G assign put
        -- Entry at coordinates (`row', `column')
    require
        valid_row: (1 <= row) and (row <= height)
        valid_column: (1 <= column) and (column <= width)
    do
        Result := array_item ((row - 1) * width + column)
    end
```

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

The multi-panel example shows the need for:

- Deferred classes and contracts (STATE, execute_state).
- Inheritance as a form of re-use and for
  - Polymorphism (states: ARRAY[STATE], current_state.execute_state)
- Focus on the data abstractions (e.g. STATE, APPLICATION) and not the "the" main functions (or "top")
  - STATE comes with its own routines e.g. display, message, etc.
- You need to think before you code:
  - Analysis classes such as ANSWER, STATE and APPLICATION only emerged after some thought

Design Pattern

- Multi-panel is a design that can be reused elsewhere
- It uses the Template and State design patterns
- For code for *state design pattern* see

https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS3311_F17

Rube Goldberg Machines