

# The State Design Pattern

Readings: OOSC2 Chapter 20



EECS3311 A: Software Design  
Fall 2019

CHEN-WEI WANG

# Motivating Problem

Consider the reservation panel of an online booking system:

-- Enquiry on Flights --

Flight sought from:  To:   
Departure on or after:  On or before:   
Preferred airline (s):  
Special requirements:

AVAILABLE FLIGHTS: 1

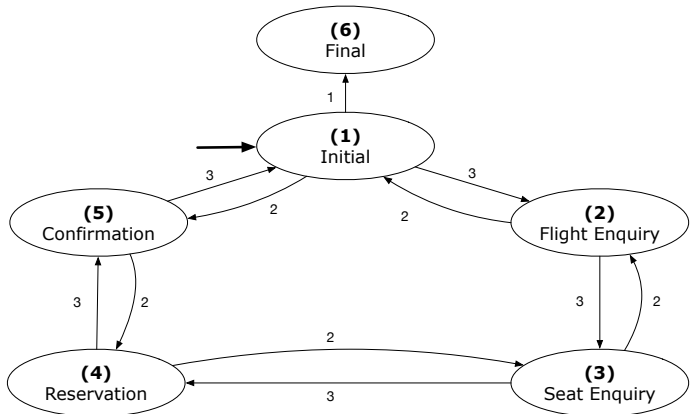
Flt#AA 42          Dep 8:25          Arr 7:45          Thru: Chicago

Choose next action:

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

# State Transition Diagram

Characterize **interactive system** as: **1)** A set of *states*; and **2)** For each state, its list of *applicable transitions* (i.e., actions).  
 e.g., Above reservation system as a **finite state machine** :



# Design Challenges

1. The state-transition graph may *large* and *sophisticated*.  
A large number  $N$  of states has  $O(N^2)$  transitions
2. The graph structure is subject to *extensions/modifications*.  
e.g., To merge “(2) Flight Enquiry” and “(3) Seat Enquiry”:  
Delete the state “(3) Seat Enquiry”.  
Delete its 4 incoming/outgoing transitions.  
e.g., Add a new state “Dietary Requirements”
3. A *general solution* is needed for such *interactive systems*.  
e.g., taobao, eBay, amazon, etc.

# A First Attempt

```
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.
```

```
3_Seat_Enquiry_panel:  
  from  
    Display Seat Enquiry Panel  
  until  
    not (wrong answer or wrong choice)  
  do  
    Read user's answer for current panel  
    Read user's choice [C] for next step  
    if wrong answer or wrong choice then  
      Output error messages  
    end  
  end  
  Process user's answer  
  case [C] in  
    2: goto 2_Flight_Enquiry_panel  
    3: goto 4_Reservation_panel  
  end
```

# A First Attempt: Good Design?

- Runtime execution  $\approx$  a **“*bowl of spaghetti*”**.
  - ⇒ The system’s behaviour is hard to predict, trace, and debug.
- **Transitions** hardwired as system’s **central control structure**.
  - ⇒ The system is vulnerable to changes/additions of states/transitions.
- All labelled blocks are largely similar in their code structures.
  - ⇒ This design **“*smells*”** due to duplicates/repetitions!
- The branching structure of the design exactly corresponds to that of the specific **transition graph**.
  - ⇒ The design is **application-specific** and **not reusable** for other interactive systems.

# A Top-Down, Hierarchical Solution

- Separation of Concern** Declare the *transition table* as a feature the system, rather than its central control structure:

```

transition (src: INTEGER; choice: INTEGER): INTEGER
    -- Return state by taking transition 'choice' from 'src' state.
    require valid_source_state: 1 ≤ src ≤ 6
               valid_choice: 1 ≤ choice ≤ 3
    ensure valid_target_state: 1 ≤ Result ≤ 6
  
```

- We may implement transition via a 2-D array.

SRC STATE \ CHOICE	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)	–	–	–

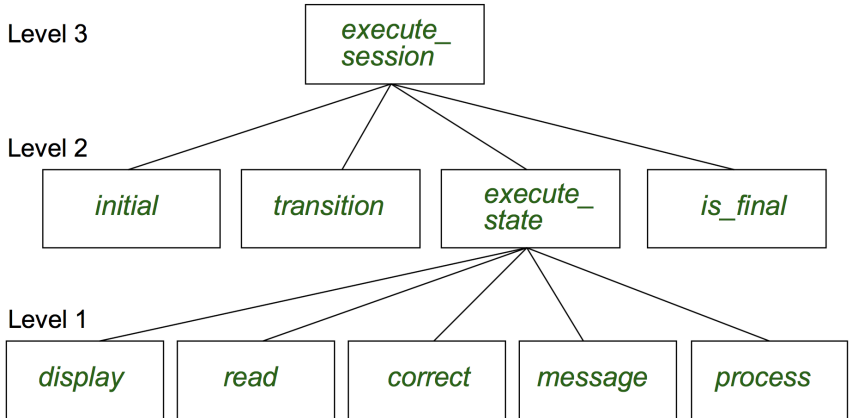
		choice		
		1	2	3
state	1	<b>6</b>	<b>5</b>	<b>2</b>
	2		<b>1</b>	<b>3</b>
	3		<b>2</b>	<b>4</b>
	4		<b>3</b>	<b>5</b>
	5		<b>4</b>	<b>1</b>
	6			

# Hierarchical Solution: Good Design?

- This is a more general solution.  
∴ *State transitions* are **separated** from the system's *central control structure*.  
⇒ **Reusable** for another interactive system by making changes only to the `transition` feature.
- How does the *central control structure* look like in this design?



# Hierarchical Solution: Top-Down Functional Decomposition



Modules of **execute\_session** and **execute\_state** are general enough on their *control structures*. ⇒ reusable

# Hierarchical Solution: System Control

All interactive sessions **share** the following *control pattern*:

- Start with some *initial state*.
- Repeatedly make *state transitions* (based on *choices* read from the user) until the state is *final* (i.e., the user wants to exit).

```
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
end
```

# Hierarchical Solution: State Handling (1)

The following *control pattern* handles all states:

```
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
```

## Hierarchical Solution: State Handling (2)

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

**Q:** How similar are the code structures of the above state-dependant commands or queries?

## Hierarchical Solution: State Handling (3)

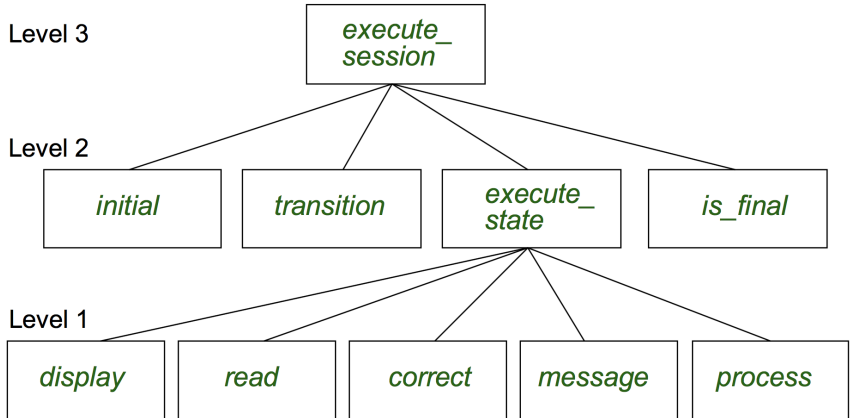
**A:** Actions of all such state-dependant features must **explicitly discriminate** on the input state argument.

```
display(current_state: INTEGER)
  require
    valid_state: 1 ≤ current_state ≤ 6
  do
    if current_state = 1 then
      -- Display Initial Panel
    elseif current_state = 2 then
      -- Display Flight Enquiry Panel
    ...
    else
      -- Display Final Panel
    end
  end
end
```

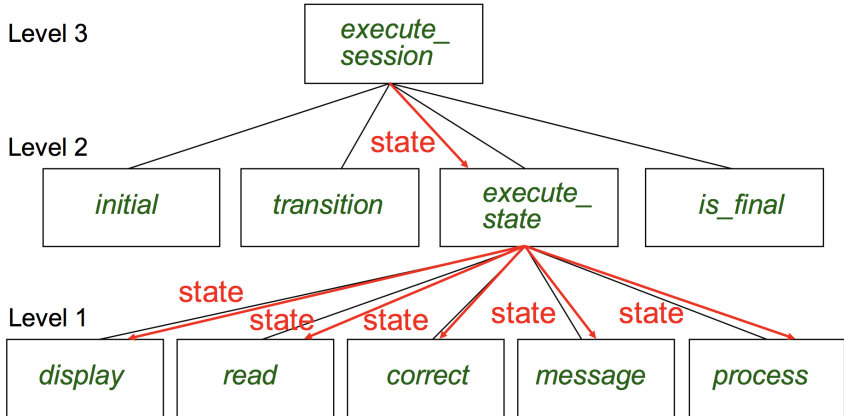
- Such design **smells** !  
∴ Same list of conditional repeats for **all** state-dependant features.
- Such design **violates** the **Single Choice Principle** .

e.g., To add/delete a state ⇒ Add/delete a branch in all such features.

# Hierarchical Solution: Visible Architecture



# Hierarchical Solution: Pervasive States



Too much data transmission: `current_state` is passed

- From `execute_session` (**Level 3**) to `execute_state` (**Level 2**)
- From `execute_state` (**Level 2**) to all features at **Level 1**

# Law of Inversion

*If your routines exchange too many data, then put your routines in your data.*

e.g.,

`execute_state` (**Level 2**) and all features at **Level 1**:

- Pass around (as *inputs*) the notion of *current\_state*
- Build upon (via *discriminations*) the notion of *current\_state*

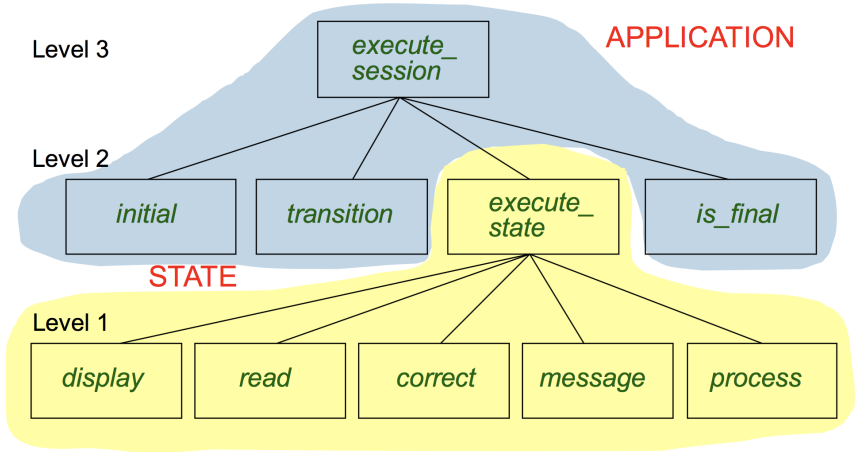
```

execute_state ( s: INTEGER )
display       ( s: INTEGER )
read_answer   ( s: INTEGER )
read_choice   ( s: INTEGER )
correct       ( s: INTEGER ; answer: ANSWER )
process       ( s: INTEGER ; answer: ANSWER )
message       ( s: INTEGER ; answer: ANSWER )
  
```

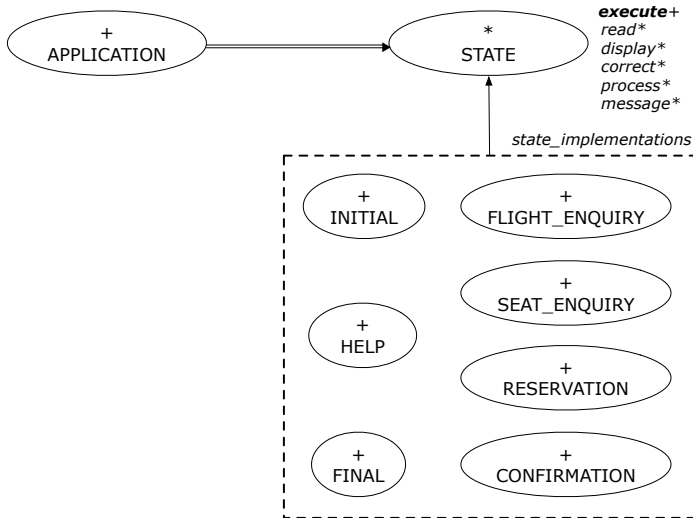
- ⇒ **Modularize** the notion of state as *class STATE*.
- ⇒ **Encapsulate** state-related information via a *STATE* interface.
- ⇒ Notion of *current\_state* becomes *implicit*: the `Current` class.



# Grouping by Data Abstractions



# Architecture of the State Pattern



# The STATE ADT

```
deferred class STATE
  read
    -- Read user's inputs
    -- Set 'answer' and 'choice'
  deferred end
  answer: ANSWER
    -- Answer for current state
  choice: INTEGER
    -- Choice for next step
  display
    -- Display current state
  deferred end
  correct: BOOLEAN
  deferred end
  process
    require correct
  deferred end
  message
    require not correct
  deferred end
```

```
execute
  local
    good: BOOLEAN
  do
    from
    until
      good
    loop
      display
      -- set answer and choice
      read
      good := correct
      if not good then
        message
      end
    end
  process
end
end
```

# The Template Design Pattern

Consider the following fragment of Eiffel code:

```

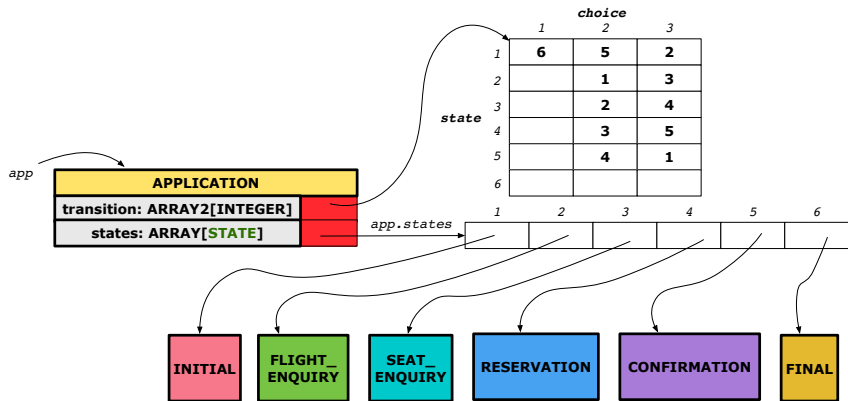
1  s: STATE
2  create {SEAT_ENQUIRY} s.make
3  s.execute
4  create {CONFIRMATION} s.make
5  s.execute
  
```

**L2** and **L4**: the same version of effective feature `execute` (from the deferred class **STATE**) is called. [ *template* ]

**L2**: specific version of effective features `display`, `process`, *etc.*, (from the effective descendant class **SEAT\_ENQUIRY**) is called. [ *template instantiated for SEAT\_ENQUIRY* ]

**L4**: specific version of effective features `display`, `process`, *etc.*, (from the effective descendant class **CONFIRMATION**) is called. [ *template instantiated for CONFIRMATION* ]

# APPLICATION Class: Array of STATE



# APPLICATION Class (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
```

## APPLICATION Class (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
```

## Example Test: Non-Interactive 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)
  -- 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
```



# APPLICATION Class (3): Interactive Session

```
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
```

# Building an Application

- Create instances of STATE.

```
s1: STATE  
create {INITIAL} s1.make
```

- Initialize an APPLICATION.

```
create app.make(number_of_states, number_of_choices)
```

- Perform polymorphic assignments on `app.states`.

```
app.put_state(initial, 1)
```

- Choose an initial state.

```
app.choose_initial(1)
```

- Build the transition table.

```
app.put_transition(6, 1, 1)
```

- Run the application.

```
app.execute_session
```

# Top-Down, Hierarchical vs. OO Solutions

- In the second (top-down, hierarchy) solution, it is required for every state-related feature to *explicitly* and *manually* discriminate on the argument value, via a list of conditionals.  
e.g., Given `display(current_state: INTEGER)`, the calls `display(1)` and `display(2)` behave differently.
- The third (OO) solution, called the State Pattern, makes such conditional *implicit* and *automatic*, by making `STATE` as a deferred class (whose descendants represent all types of states), and by delegating such conditional actions to *dynamic binding*.  
e.g., Given `s: STATE`, behaviour of the call `s.display` depends on the *dynamic type* of `s` (such as `INITIAL` vs. `FLIGHT_ENQUIRY`).

# Index (1)

---

**Motivating Problem**

**State Transition Diagram**

**Design Challenges**

**A First Attempt**

**A First Attempt: Good Design?**

**A Top-Down, Hierarchical Solution**

**Hierarchical Solution: Good Design?**

**Hierarchical Solution:**

**Top-Down Functional Decomposition**

**Hierarchical Solution: System Control**

**Hierarchical Solution: State Handling (1)**

**Hierarchical Solution: State Handling (2)**

**Hierarchical Solution: State Handling (3)**

**Hierarchical Solution: Visible Architecture**

## **Index (2)**

---

**Hierarchical Solution: Pervasive States**

**Law of Inversion**

**Grouping by Data Abstractions**

**Architecture of the State Pattern**

**The STATE ADT**

**The Template Design Pattern**

**APPLICATION Class: Array of STATE**

**APPLICATION Class (1)**

**APPLICATION Class (2)**

**Example Test: Non-Interactive Session**

**APPLICATION Class (3): Interactive Session**

**Building an Application**

**Top-Down, Hierarchical vs. OO Solutions**