Lecture 8

Part 1

General Book: Storage vs. Retrieval

General Book

Supplier

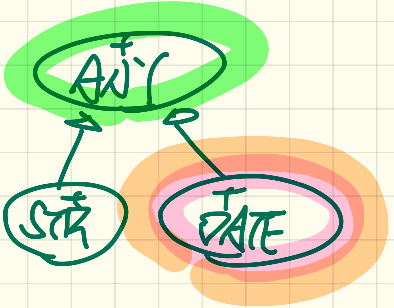
```
class BOOK
  names: ARRAY[STRING]
  records: ARRAY[ANY]
  -- Create an empty book
  make do ... end
  -- Add a name-record pair to the book
  add (name: STRING; record: ANY) do ... end
  -- Return the record associated with a given name
  get (name: STRING): ANY do ... end
end
```

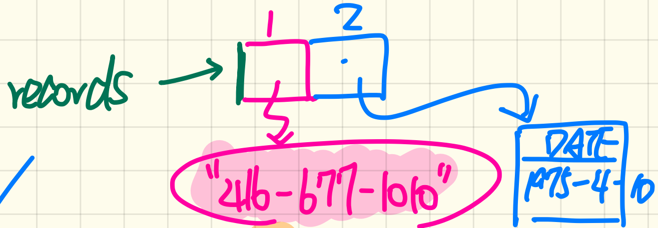
Client

```
birthday: DATE; phone_number: STRING
b: BOOK; is_wednesday: BOOLEAN
create {BOOK} b.make
phone_number := "416-677-1010"
b.add ("SuYeon", phone_number)
create {DATE} birthday.make(1975, 4, 10)
b.add ("Yuna", birthday)
is_wednesday := b.get("Yuna").get_day_of_week = 4
```

General Book: Retrieval from Polymorphic Arrary

```
birthday: DATE; phone_number: STRING
b: BOOK; is_wednesday: BOOLEAN
create {BOOK} b.make
phone_number := "416-677-1010"
b.add ("SuYeon", phone_number)
create {DATE} birthday.make(1975, 4, 10)
b.add ("Yuna", birthday)
```





	ST	DT
records[1]	ANY	STRING
records[2]		DATE

✓

```
check attached {DATE} b.get ("Yuna") as yuna_bday then
  is_wednesday := yuna_bday.get_day_of_week = 4
end
```

↳ ST : DATE.

```
check attached {DATE} b.get ("SuYeon") as suyeon_bday then
  is_wednesday := suyeon_bday.get_day_of_week = 4
end
```

F.

⇒ Violation of R.T. expectation on cast type DATE

→ downward cast but DT STR can't fulfil

General Book violates Single Choice Principle

Storage

```
rec1: C1
... -- declarations of rec2 to rec99
rec100: C100
create {C1} rec1.make(...) ; b.add(..., rec1)
... -- additions of rec2 to rec99
create {C100} rec100.make(...) ; b.add(..., rec100)
```

Retrievals

```
-- assumption: 'f1' specific to C1, 'f2' specific to C2, etc.
if attached {C1} b.get("Jim") as c1 then
  c1.f1
... -- cases for C2 to C99
elseif attached {C100} b.get("Jim") as c100 then
  c100.f100
end
```

elseif attached {C101} -- then --

```
-- assumption: 'f1' specific to C1, 'f2' specific to C2, etc.
if attached {C1} b.get("Jim") as c1 then
  c1.f1
... -- cases for C2 to C99
elseif attached {C100} b.get("Jim") as c100 then
  c100.f100
end
```

elseif attached {C101} ... then --

disadvantage

repetition of check structure ⇒ violation of SCP.

What if a new type **C101** is introduced? X

What if type **C100** becomes obselete?

Lecture 8

Part 2

Generic Book: Storage vs. Retrieval

Generic Book

Supplier

```
class BOOK[G]
  names: ARRAY[STRING]
  records: ARRAY[G]
  -- Create an empty book
  make do ... end
  /* Add a name-record pair to the book */
  add (name: STRING; record: G) do ... end
  /* Return the record associated with a given name */
  get (name: STRING): G do ... end
end
```

Client

```
birthday: DATE; phone_number: STRING
b: BOOK[DATE] ; is_wednesday: BOOLEAN
create {BOOK[DATE]} b.make
phone_number = "416-67-1010"
b.add ("SuYeon", phone_number)
create {DATE} birthday.make (1975, 4, 10)
b.add ("Yuna", birthday)
is_wednesday := b.get("Yuna").get_day_of_week == 4
```

Instantiating Generic Parameters

Say the **supplier** provides a generic `DICTIONARY` class:

```
class DICTIONARY[V, K] -- V type of values; K type of keys
  add_entry (v: V; k: K) do ... end
  remove_entry (k: K) do ... end
end
```

Clients use `DICTIONARY` with different degrees of instantiations:

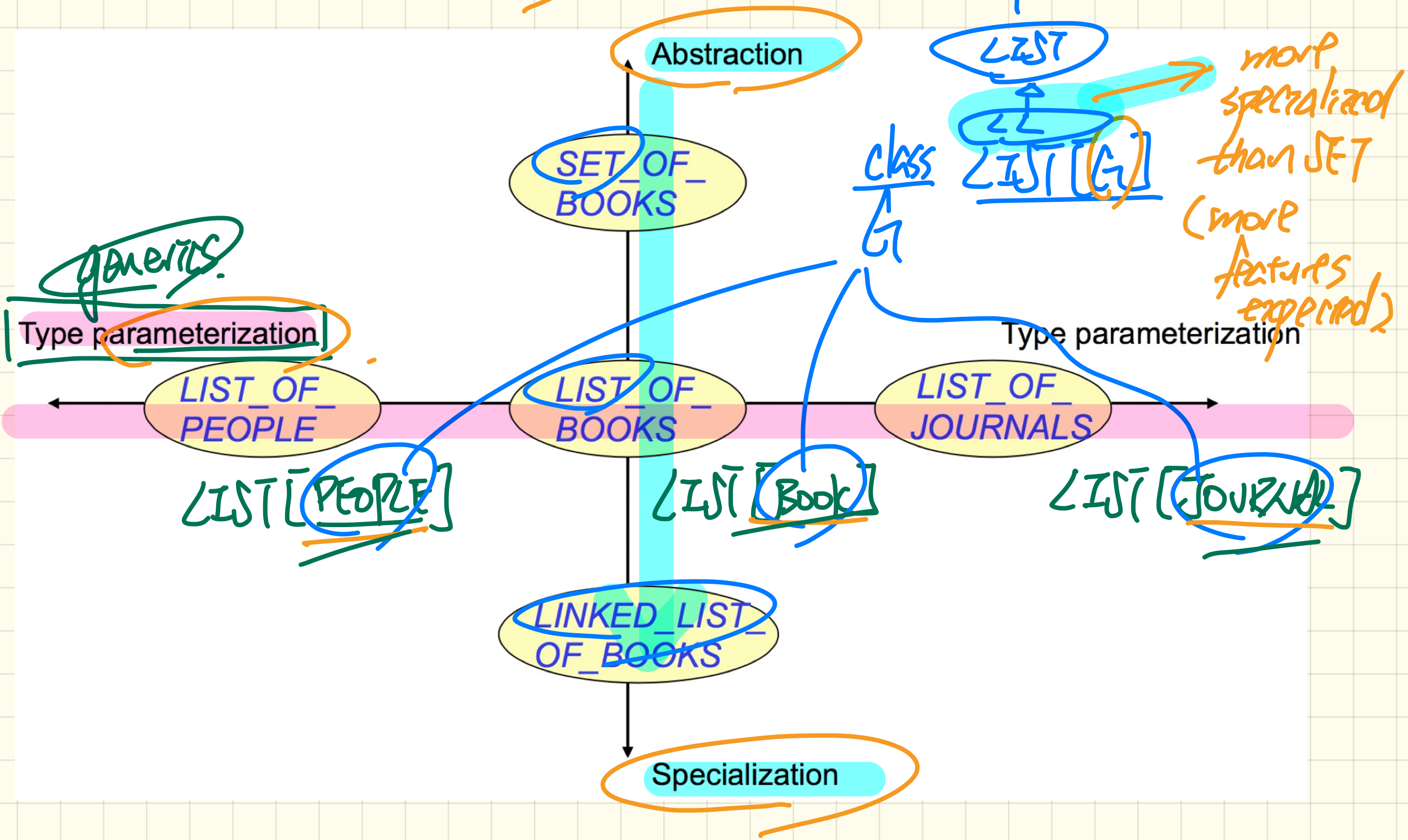
```
class DATABASE_TABLE[K, V]
  imp: DICTIONARY[V, K]
end
```

e.g., Declaring `DATABSE_TABLE[INTEGER, STRING]` instantiates `DICTIONARY[STRING, INTEGER]`.

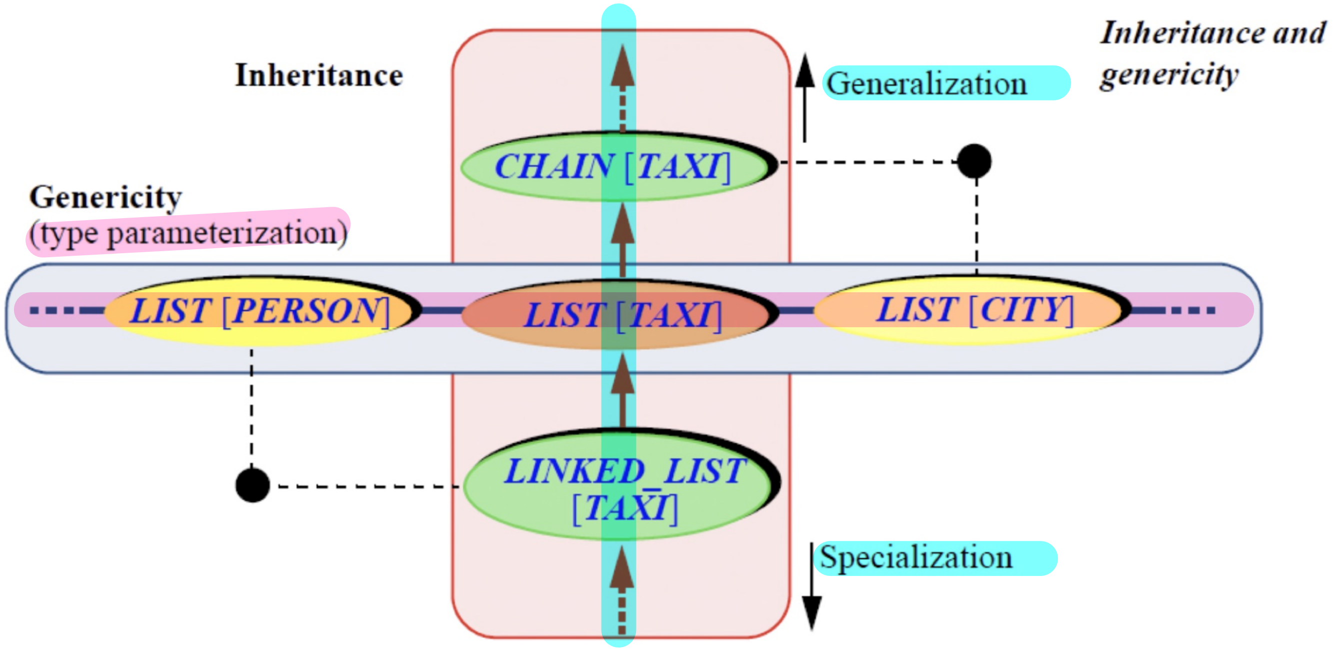
```
class STUDENT_BOOK[V]
  imp: DICTIONARY[V, STRING]
end
```

e.g., Declaring `STUDENT_BOOK[ARRAY[COURSE]]` instantiates `DICTIONARY[ARRAY[COURSE], STRING]`.

Generics vs. Inheritance (1)



Generics vs. Inheritance (2)



Lecture 8

Part 3

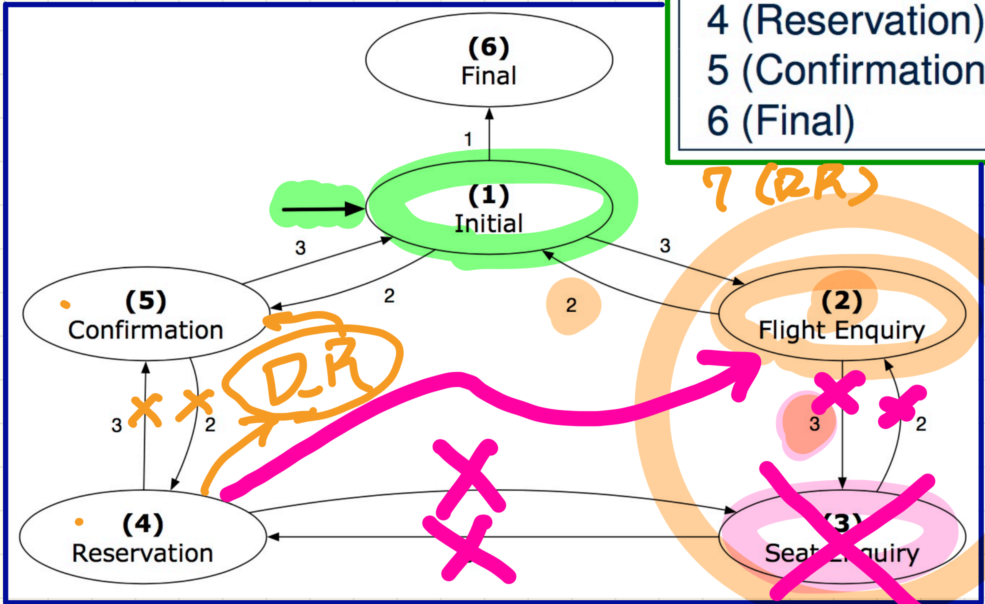
Motivating Problem: Interactive Systems

Finite State Machine (FSM)

State Transition Table

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)	–	–	–

State Transition Diagram

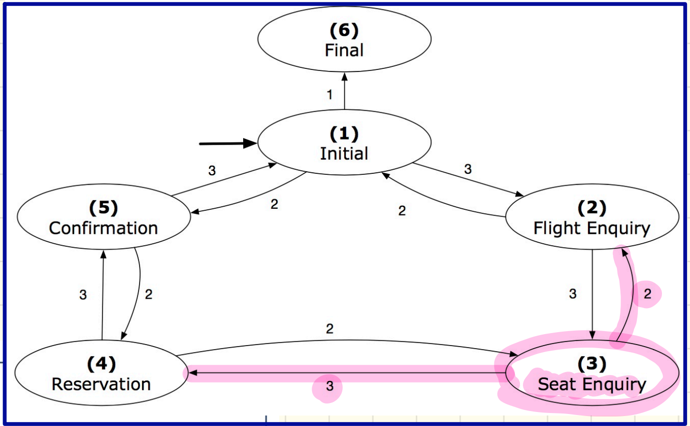


Lecture 8

Part 4

First Design: Assembly Style

Design of a Reservation System: 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
```

<u>Lecture 8</u>

<u>Part 5</u>

Second Design: Hierarchical Style

Design of a Reservation System: Second Attempt (1)

```
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
```

Examples:

transition(3, 2) → 2 (target state)

transition(3, 3) → 4

State Transition Table

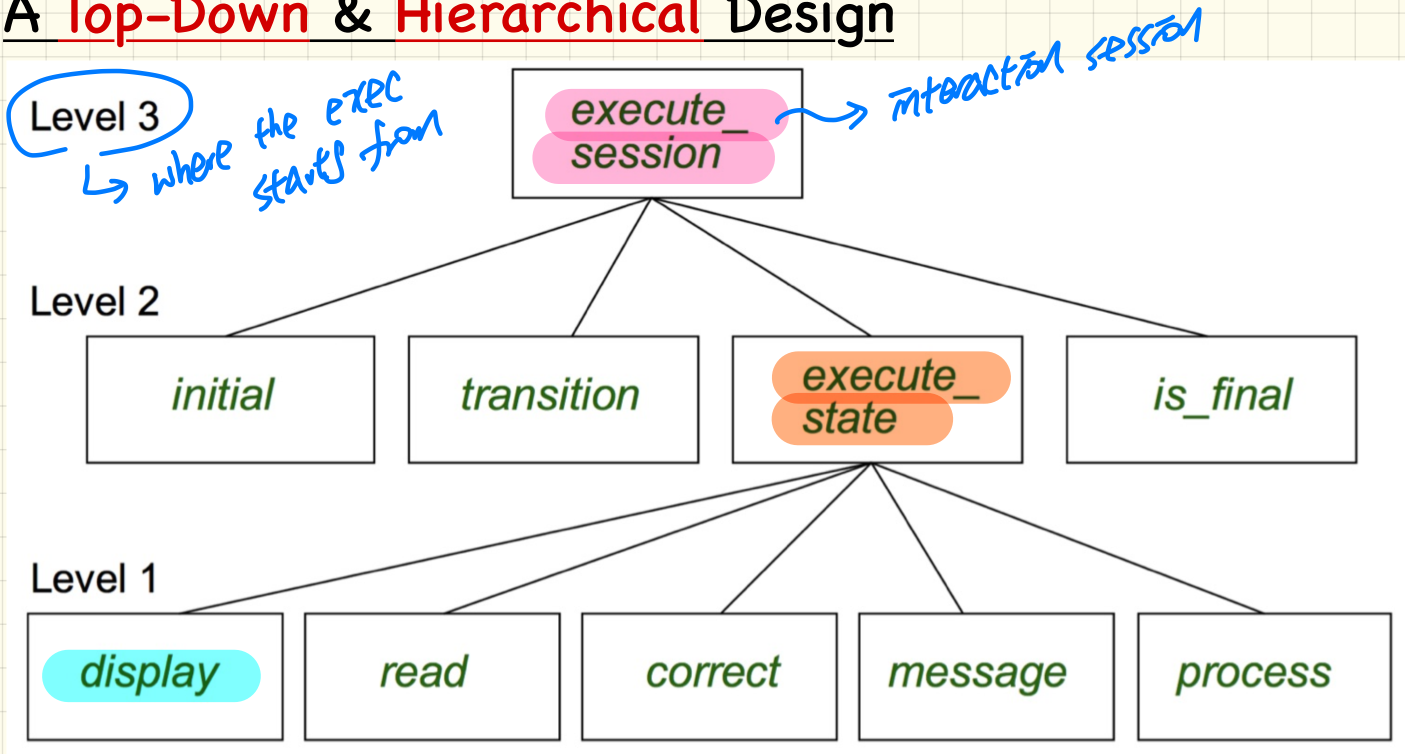
Src State \ Choice	1	2	3
1 (Initial)	6	5	2
2 (Flight Enquiry)	–	1	3
3 (Seat Enquiry)	–	2	4
4 (Reservation)	–	3	5
5 (Confirmation)	–	4	1
6 (Final)	–	–	–

2D Array Implementation

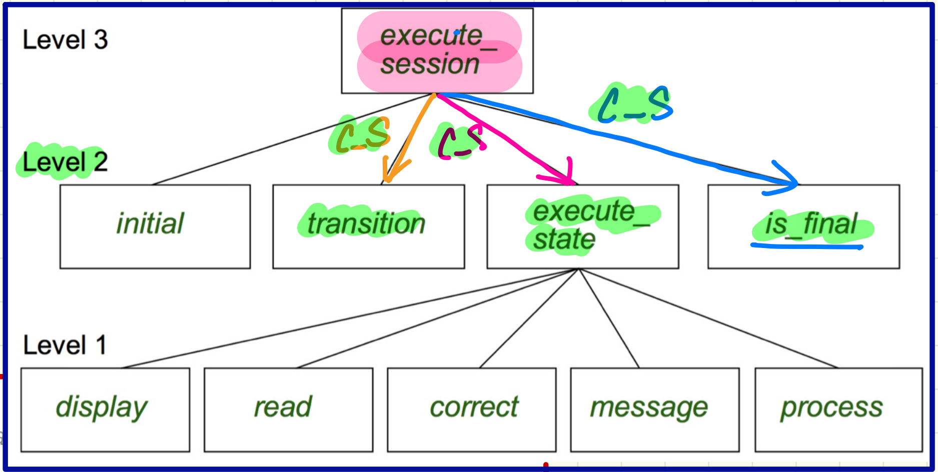
state \ choice	1	2	3
1	6	5	2
2		1	3
3		2	4
4		3	5
5		4	1
6			

Design of a Reservation System: Second Attempt (2)

A Top-Down & Hierarchical Design



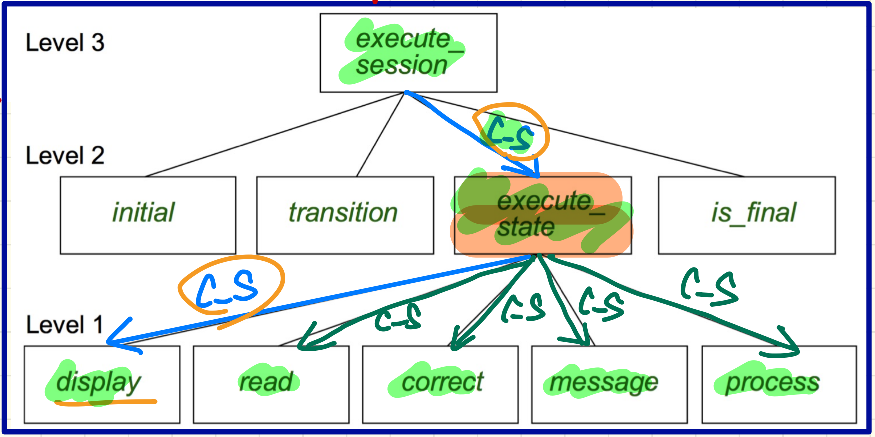
Design of a Reservation System: Second Attempt (3)




```
execute_session
    -- Execute a full intera
  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
```

Design of a Reservation System: Second Attempt (4)

```
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
```

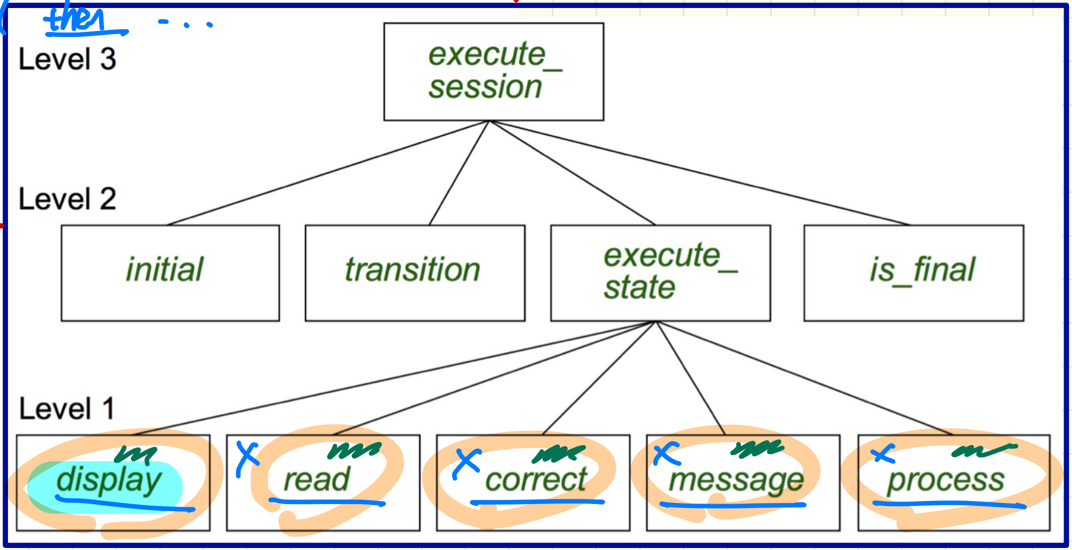


Design of a Reservation System: Second Attempt (5)

```
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 ...
    end
  end
```

Add a new state?

Delete an existing state?



Lecture 8

Part 6

Template & State Patterns: Supplier

Moving from Top-Down Design to OO Design

Context object.

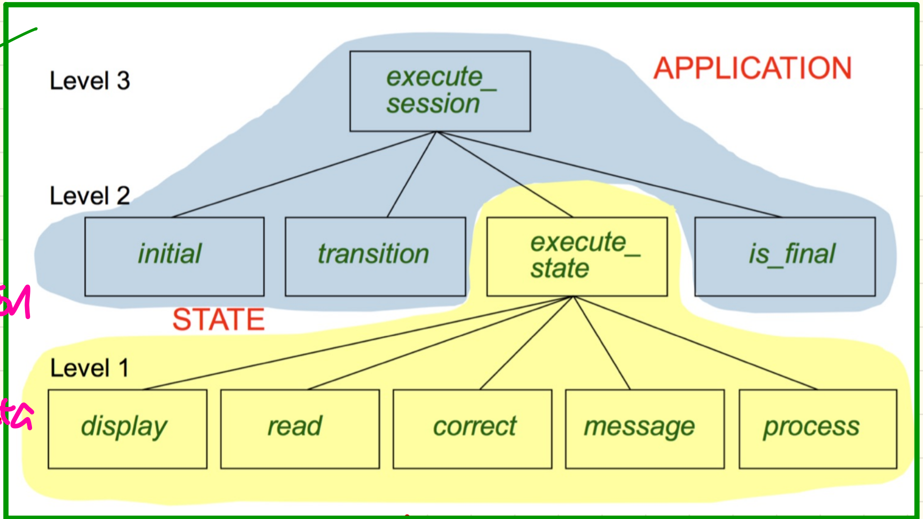
Object-Oriented

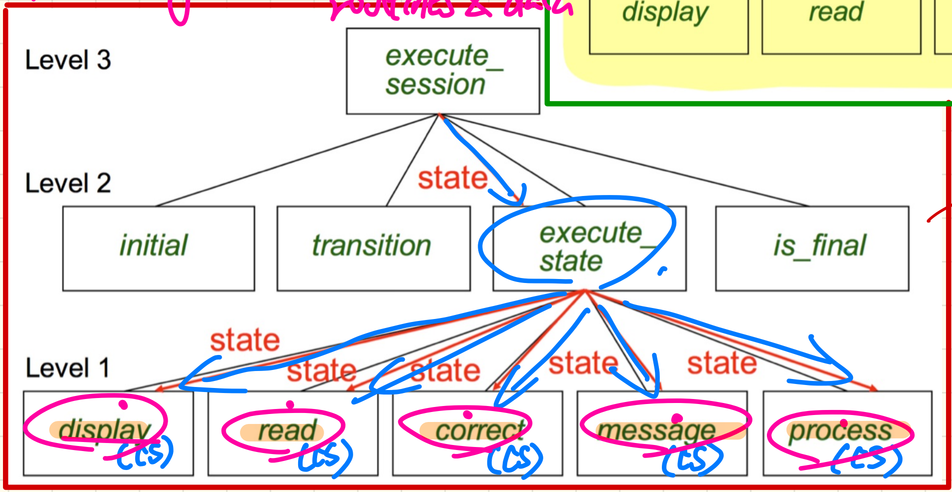
current_state: STATE
current_state.execute

class → abstraction → state-related routines & data

parameterless.

polymorphism → dynamic binding.

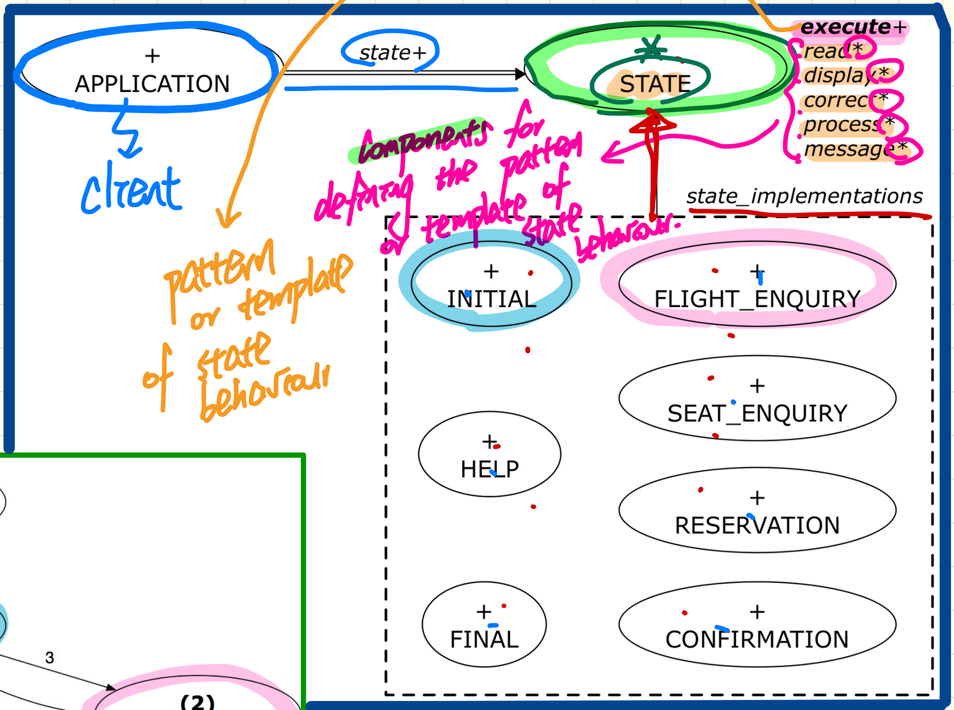


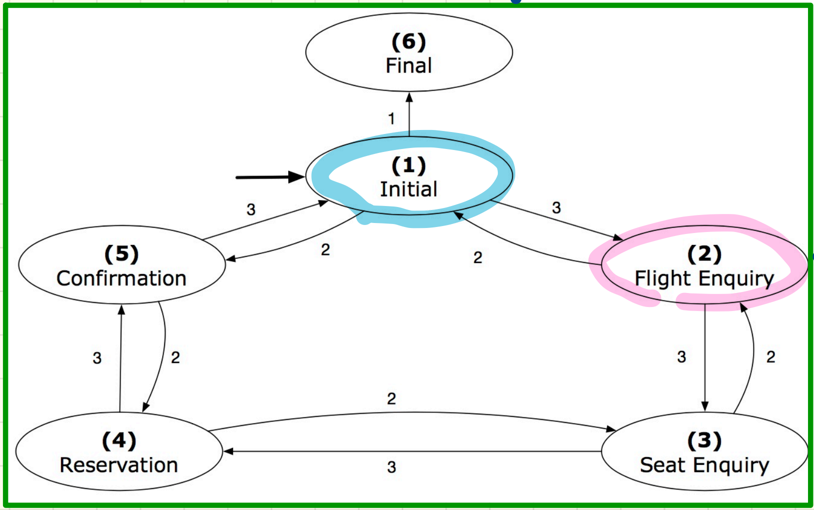


Top-Down

current_state: INTEGER
execute_state(current_stste)

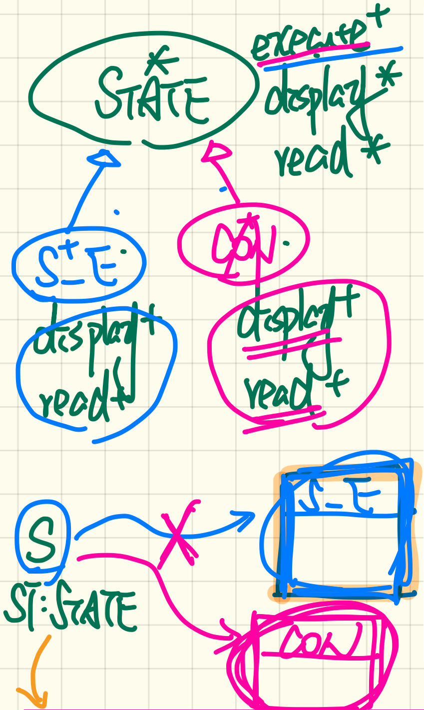
State Pattern: Architecture






```
s: STATE
create {SEAT_ENQUIRY} s.make
s.execute   → w.r.t. DT
create {CONFIRMATION} s.make
s.execute   → w.r.t. DT
```

State Pattern: State Module




```
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
```

effective (implemented)

```
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
```

TEMPLATE

(pattern)

```
s: STATE
create {SEAT_ENQUIRY} s.make
s.execute
create {CONFIRMATION} s.make
s.execute
```

{STATE}.execute → {S_E}.display, {S_E}.read

{STATE}.execute → {CON}.display, {CON}.read

Lecture 8

Part 6

Template & State Patterns: Client

State Pattern: Application Module

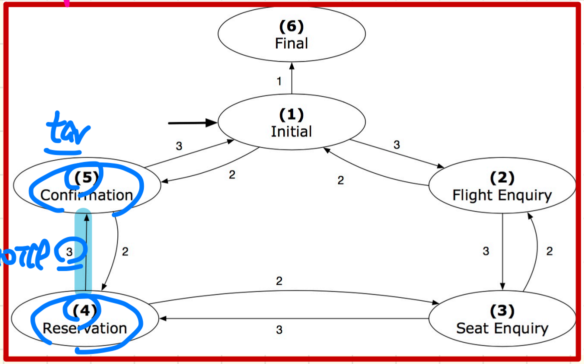
```
class APPLICATION create make
feature {NONE} -- Implementation of Transition Graph
  transition: ARRAY2[INTEGER]
    -- State transitions: transition[state, choice]
  states: ARRAY[STATE]
    -- State for each index, constrained by size of 'transition'
feature
  initial: INTEGER
  number_of_states: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

call by value
put_state(@, ...)

a polymorphic collection of states

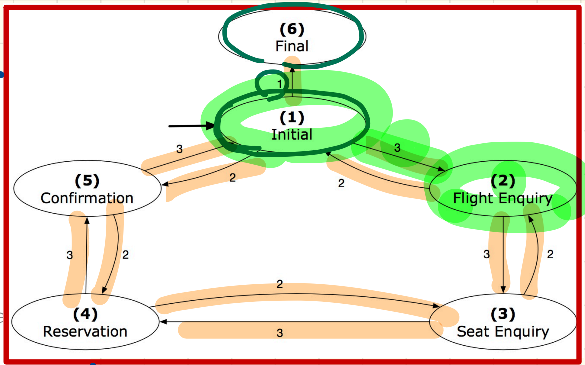
consequence of having a polymorphic argument is we have

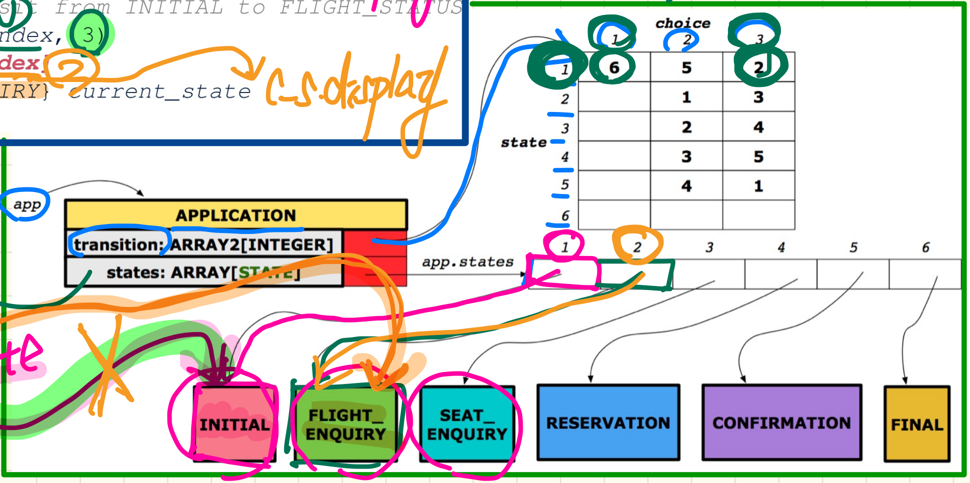
put_tran(5, 4, 3)



State Pattern: Test

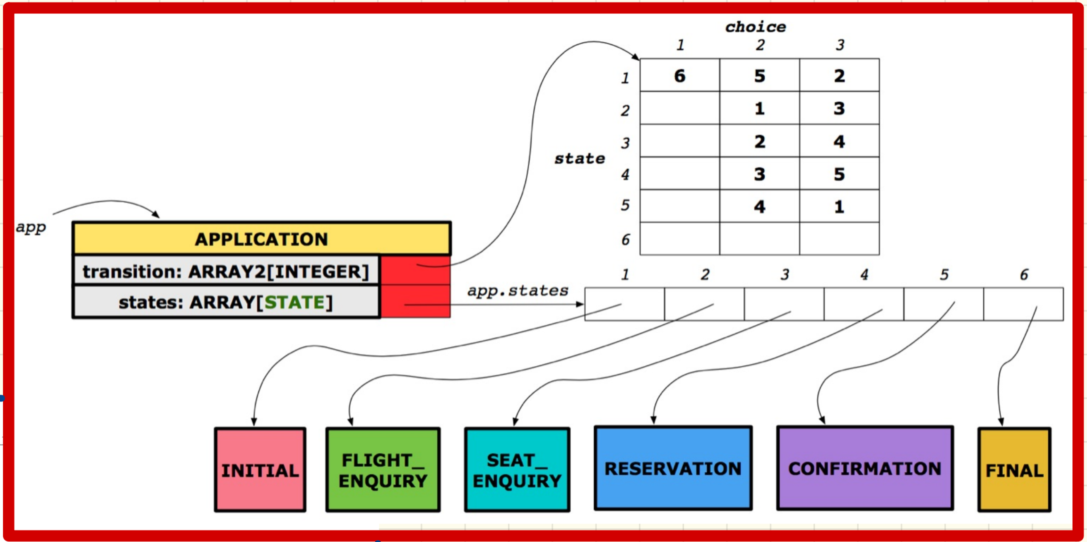
```
test_application: BOOLEAN
  local
    app: APPLICATION ; current_state: STATE ; index: INTEGER
  do
    create app.make (6, 3)
    app.put_state (create {INITIAL}.make, 1)
    -- Similarly for other 5 states.
    app.choose_initial (1)
    -- Transit to FINAL given current state INITIAL and choice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





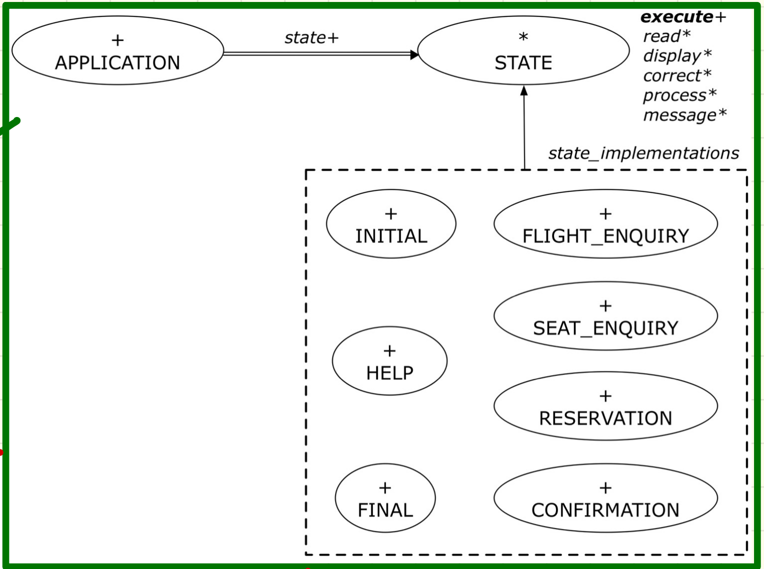
state \ choice	1	2	3
1	6	5	2
2		1	3
3		2	4
4		3	5
5		4	1
6			

State Pattern: Interactive Session



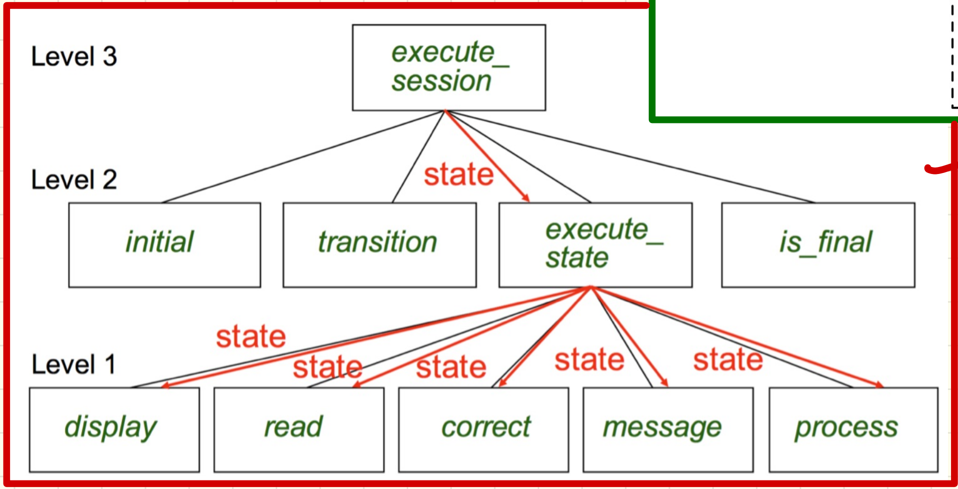

```
class APPLICATION
feature {NONE} -- Implementat
  transition: ARRAY2[INTEGER]
  states: ARRAY[STATE]
feature
  execute_session
    local
      current_state: STATE
      index: INTEGER
    do
      from
        index := initial
      until
        is_final (index)
      loop
        current_state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

Interactive System: Top-Down Design vs. OO Design



Object-Oriented

current_state: **STATE**
current_state.*execute*



Top-Down

current_state: INTEGER
execute_state(current_stste)