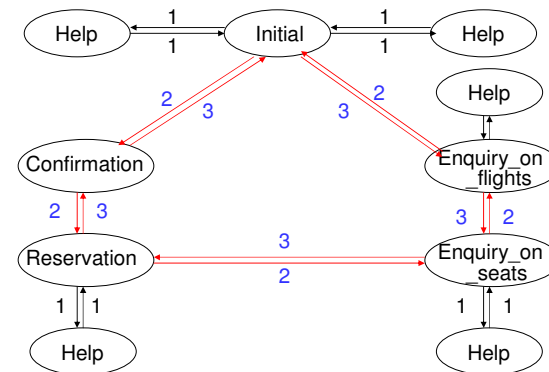


## An O-O design example

## The transition diagram



## A reservation panel

-- 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

## A first attempt

```

PEnquiry_on_flights:
  output "enquiry on flights" screen
  repeat
    read user's answers and his exit choice C
    if error in answer then
      output message
    end
  until no error in answer
  end
  process answer
  inspect C
  when C0 then
    goto Exit
  when C1 then
    goto PHelp
  ...
  when Cn-1 then
    goto PReservation
  end
  ...
    
```

(and similarly for each state)

## What's wrong with the previous scheme?

- Intricate branching structure ("spaghetti bowl").
- Extensibility problems: dialogue structure wired into program structure.

5



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## The transition function

	0	1	2	3
0 (Initial)			2	
1 (Help)	Exit	Return		
2 (Conf.)	Exit		3	0
3 (Reserv.)	Exit		4	2
4 (Seats)	Exit		5	3
5 (flights)	Exit		0	4

7



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## A functional, top-down solution

For more flexibility, represent the structure of the transition diagram by a function

$transition(i, k)$

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.

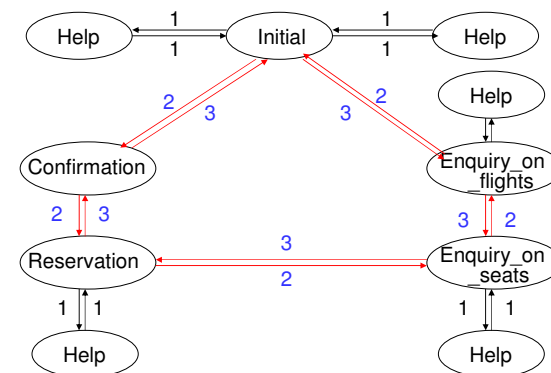
6



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## The transition diagram



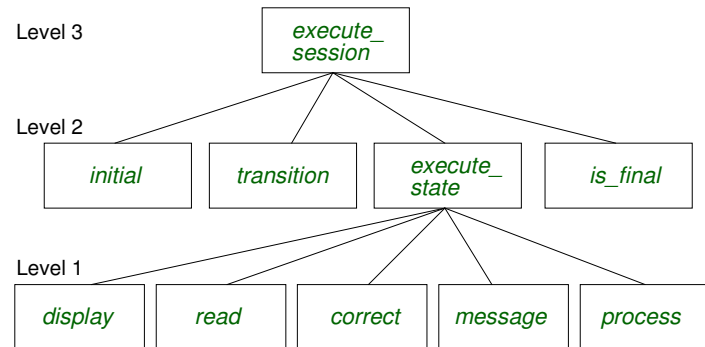
8



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## New system architecture



9



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## To describe an application

- Provide *transition* function
- Define *initial* state
- Define *is\_final* function

11



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## New system architecture

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 is
  -- 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
```

10



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Actions in a state

```
execute_state(current_state: INTEGER): INTEGER is
  -- Actions for current_state, returning 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)
  return
  choice
end
```

12



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## 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.

13



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## 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)
```

15



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## 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)
```

14



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Data transmission

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

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

16



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## 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 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.

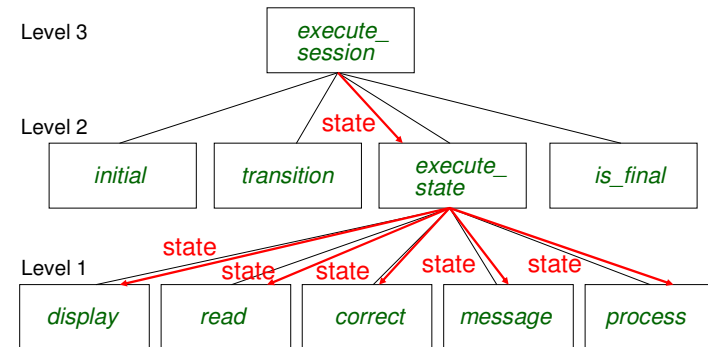
17



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## The real story



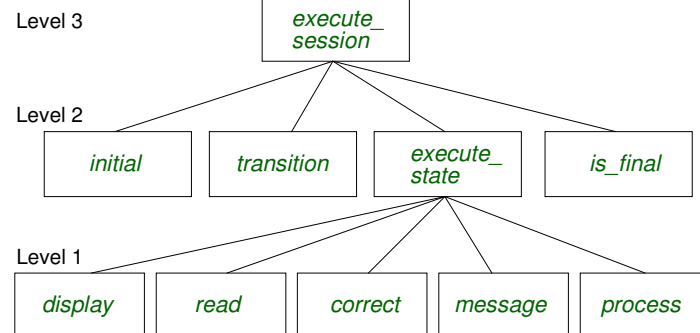
19



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## The visible architecture



18



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## The law of inversion

The everywhere lurking state

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

20



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## 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.

21



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Class *STATE*

```
deferred class
  STATE
feature
  choice: INTEGER -- User's selection for next step

  input: ANSWER -- User's answer for this step

  display is
    deferred
    end -- Show screen for this step.

  read is
    -- Get user's answer and exit choice,
    -- recording them into input and choice.

    deferred
    ensure
      input /= Void
    end

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

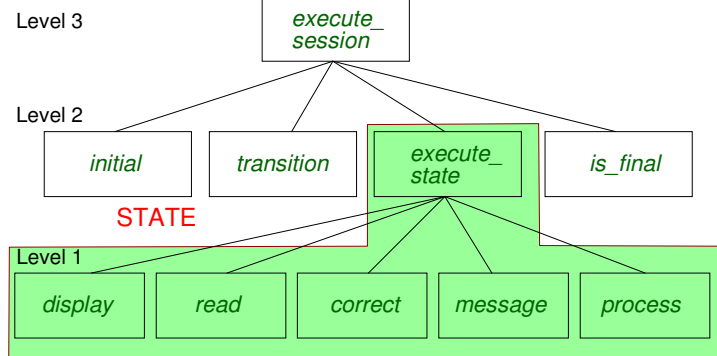
23



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Grouping by data abstractions



22



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Class *STATE*

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

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

24



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Class *STATE*

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

25



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## To describe a state of an application

Introduce new descendant of *STATE*:

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

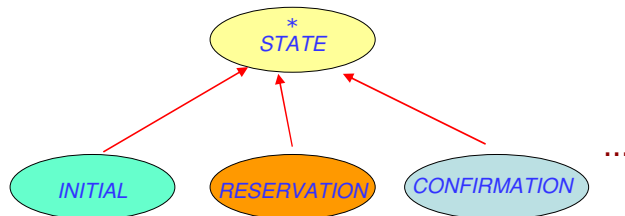
27



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Class structure



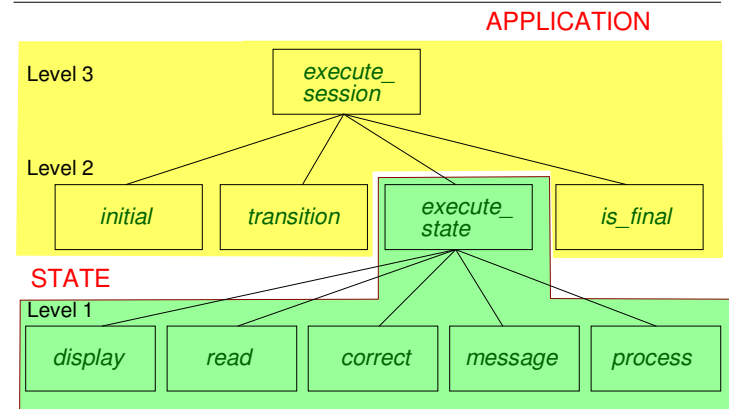
26



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Rearranging the modules



28



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## 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*.

29



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Describing an application

```
class
  APPLICATION
create
  make
feature
  initial: INTEGER
  make (n, m: INTEGER) is
    -- 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
```

31



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## 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*.

30



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Array of states: A polymorphic container

```
states: ARRAY [STATE]
```

Notations for accessing array element,

i.e. *states [i]* in Pascal:

```
states.item (i)
states @ i
```

(Soon in Eiffel: just *states [i]*)

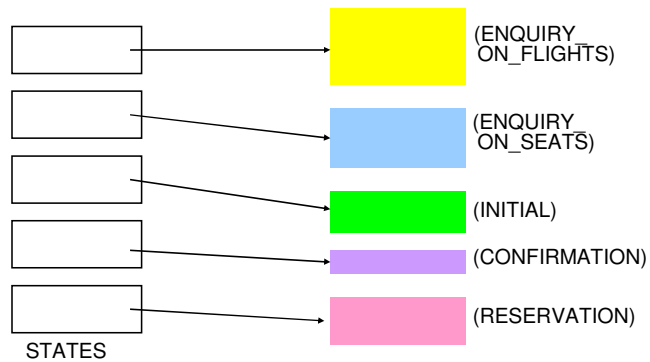
32



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## The array of states



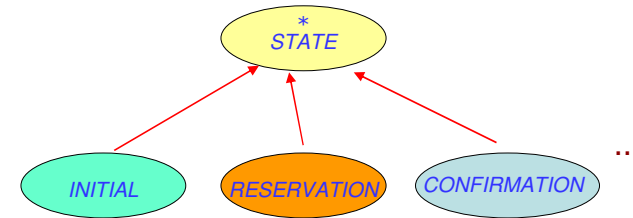
33



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Class structure



35



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Executing a session

```

execute_session is
  -- Run one session of application
  local
    current_state: STATE -- Polymorphic!
    index: INTEGER
  do
    from
      index := initial
    invariant
      0 <= index
      index <= n
    until
      index = 0
    loop
      current_state := states @ index
      current_state.execute_state
      check
        1 <= current_state.choice
        current_state.choice <= m
      end
      index := transition.item(index, current_state.choice)
    end
  end
end
  
```

34



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## Other features of APPLICATION

```

put_state(s: STATE; number: INTEGER) is
  -- Enter state s with index number.
  require
    1 <= number
    number <= states.upper
  do
    states.put(number, s)
  end

choose_initial(number: INTEGER) is
  -- Define state number number as the initial
  -- state.
  require
    1 <= number
    number <= states.upper
  do
    first_number := number
  end
  
```

36



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## More features of *APPLICATION*

```
put_transition(source, target, label: INTEGER) is
  -- 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
```

37



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## 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
- ...

39



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## 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 *no*:

```
a.choose_initial(no)
```

Enter transitions:

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

May now run:

```
a.execute_session
```

38



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## 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.

40



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

## 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

41



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH

End of lecture 18



Chair of Software Engineering  
ETH

## Object-Oriented Design

It's all about finding the right data abstractions

42



Object-Oriented Software Construction, 2005

Chair of Software Engineering  
ETH