Case Study
Multi-Panel Interactive System

The Problem Domain

- Build a general type of interactive system
  - » **Users interact with a set of panels**
    - > **Web applications are an example**
- Each session goes through a number of states
    - > **Finite state machine**
    - > **Automatic Teller Machine**
  - » **A state corresponds to a fill-in-the-blanks panel**
    - > **User is adding to a database of information**
  - » **Depending upon user choices transitions occur to other states**

Example Panel

– Enquiry on Flights –

Flight from Somewhere Flight to Anywhere

Departure on or after not soon enough

on or before too late

Preferred airline(s):

Special requirements: ______________________

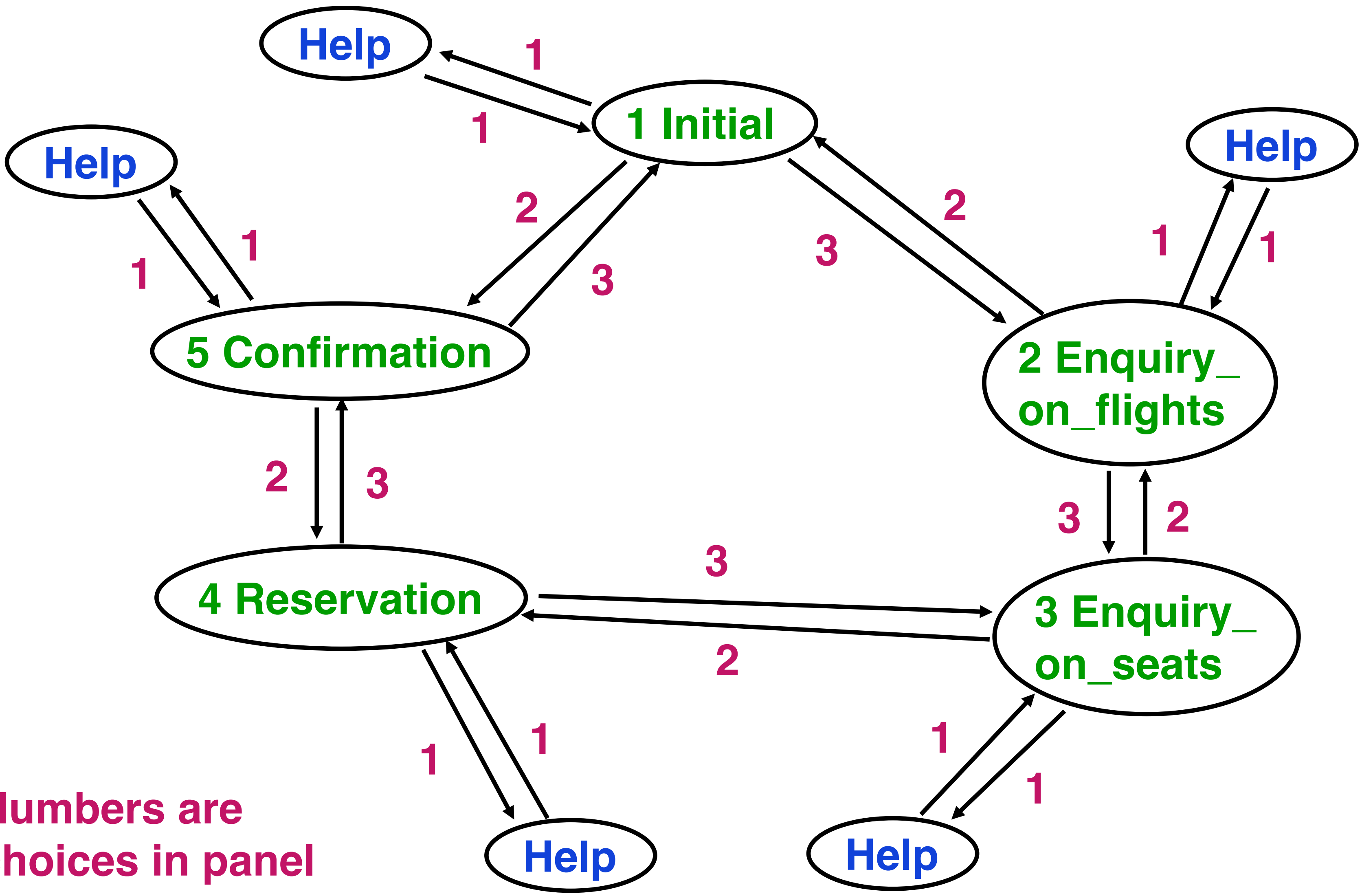
Available flights: 1

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

Choose next_action

0 Exit 1 Help 2 Further enquiry 3 Reserve seat

A State Transition Diagram



Numbers are choices in panel

General Design Goals

- Create a design and implementation for such applications
- General & flexible solution
- Things to think about
  - » **Finite state machine may be very large**
    - > **Applications can have hundreds of states and thousands of transitions**
  - » **Structure of the system is subject to change**
    - > **Cannot foresee all possible states & transitions**
  - » **No specific application is mentioned**
    - > **What if you need many variations**

First Attempt

- Block/Module oriented – procedural
- System made of a number of blocks
  - » **One for each state in the FSM**
    - » **Follows the Direct Mapping Rule**

First Attempt – 2

```
Enquiry_Block
    display panel
    repeat
        get user's answer and choice C for next step
        if error in answer then output error fi
    until not error in answer

   "Process answer"
   case C in
       C0 : goto Exit_Block
       C1 : goto Help_Block
       C2 : goto Reservation_Block
       ...
   esac
```

Similarly for all other states

Easy to devise, does the job

Terrible for meeting requirements

First Attempt – 3

```
Enquiry_Block
    display panel
    repeat
        get user's answer and choice C for next step
        if error in answer then output error fi
    until not error in answer

   "Process answer"
   case C in
       C0 : goto Exit_Block
       C1 : goto Help_Block
       C2 : goto Reservation_Block
       ...
   esac
```

What are the problems?

Block Design Problems

- Use **goto**'s (Dijkstra)
  - » **Usually symptomatic of deeper problem**
- Branch structure (**goto**'s) are an exact implementation of the graph
  - » **Vulnerable to change**
    - > **Add a new state**
      - – **add new block, change all other blocks**
    - > **Add a new transition**
      - – **Change all blocks that should use it**

Block Design Problems – 2

- Forget **reusability** across applications
  - » **Specific to one application**
- Want not just a solution but a **quality solution**
  - » **Have to work harder**
- **What does quality mean for this system?**

Quality Design

- A general design – a set of reusable modules – would be a huge benefit

Quality Design – 2

- A general design – a set of reusable modules – would be a huge benefit

- **Getting the problem to work is only a part of the solution and insufficient for the task**

Quality Design – 3

- A general design – a set of reusable modules – would be a huge benefit
- Getting the problem to work is only a part of the solution and insufficient for the task
- **Customer's requirements go far beyond**
  - » **mere correctness**
  - » **mere functionality**

FSM Representation

- Problems seems to be due to the traversal (**goto**) structure
  - **The representation of the finite-state machine**

FSM Representation – 2

- Problems seems to be due to the traversal (**goto**) structure
  - **The representation of the finite-state machine**

What can we do?

FSM Representation – 3

- **Generalizing the transition diagram will gain generality**

- **???**

FSM Representation – 4

- Generalizing the transition diagram will gain generality

- **Model the function transition as a transition table representation of a FSM**
  - » **Designate one state as initial**
  - » **One or more states as final**

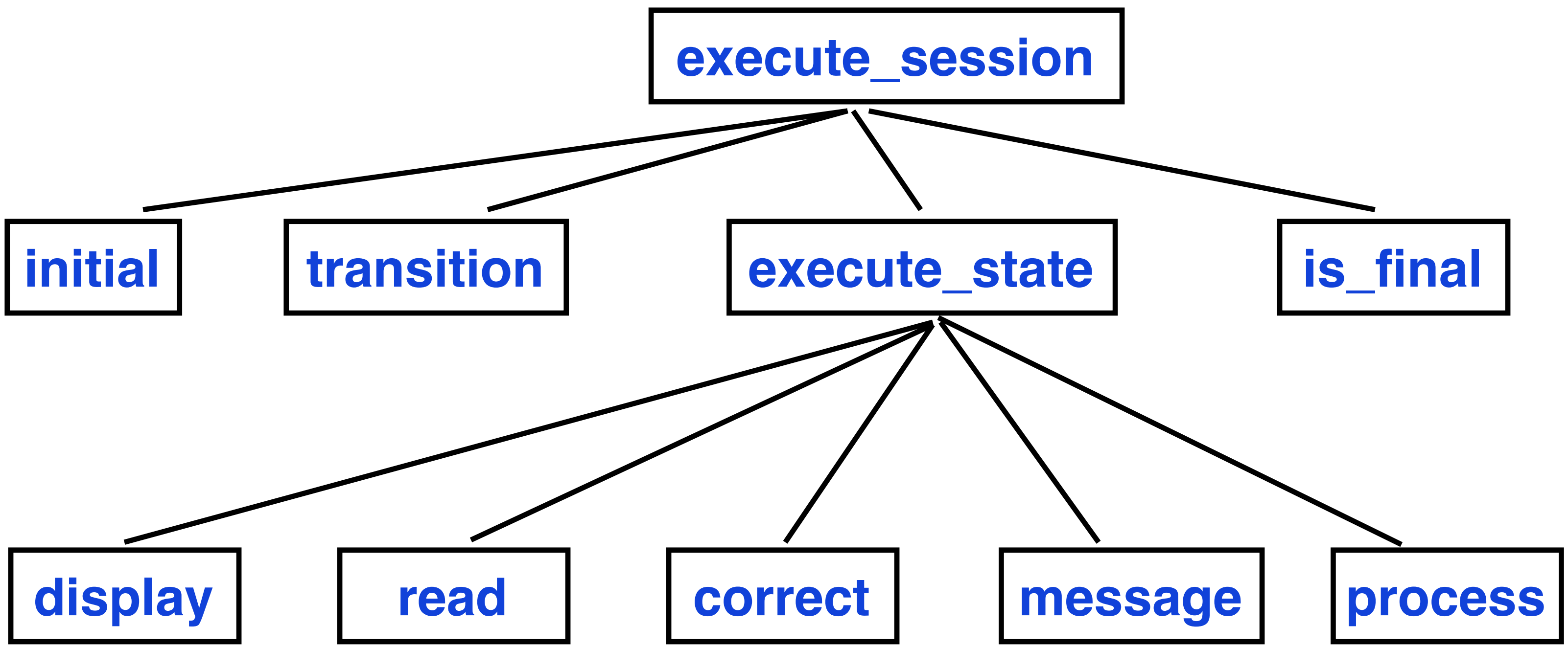
Transition Table

Choice

State

	0	1	2	3
1 Initial	-1	0	5	2
2 Flights		0	1	3
3 Seats		0	2	4
4 Reserv.		0	3	5
5 Confirm		0	4	1
0 Help		back		
-1 Final				

Top Down Decomposition



Implement execute_session

```
execute_session
    -- Execute a complete session
    local state, next : INTEGER
    do
      state := initial  -- start in initial state
      repeat
                                              -- next is a VAR parameter
         execute_state (state, next )
         state := transition ( state, next )
      until is_final ( state )
      end
    end
```

Implement execute_state

```
execute_state ( in s : INTEGER , out c : INTEGER )
    -- c contains the user's choice for next state
    local a : ANSWER  ;  ok :BOOLEAN
    do
      repeat
         display (s )     -- display panel for state s
         read ( s , a )   -- get user answer in a
         ok := correct ( s , a )
      until ok end
      process ( s , a )
      c := next_choice ( a )    -- get user choice for panel
    end
```

What are the problems?

Implement execute_state – 2

```
execute_state ( in s : INTEGER , out c : INTEGER )
    -- c contains the user's choice for next state
    local a : ANSWER  ;  ok :BOOLEAN
    do
      repeat
         display (s )     -- display panel for state s
         read ( s , a )   -- get user answer in a
         ok := correct ( s , a )
      until ok end
      process ( s , a )
      c := next_choice ( a )    -- get user choice for panel
    end
```

State s is argument for all functions!

What will be the structure/design of display?

Top Down Problems?

- **Tight coupling**
  - » **State is argument to every routine**

Top Down Problems? – 2

- Tight coupling
  - » State is argument to every routine
- **Means long and complicate control structure**
  - » **Case statements everywhere on state**

Top Down Problems? – 3

- Tight coupling
  - » State is argument to every routine
- Means long and complicate control structure
  - » Case statements everywhere on state
- **Violates single choice principle**
  - » **Too many locations need to know about all states**
    - > **Difficult to modify as states added or removed**

Top Down Problems? – 4

- Tight coupling
  - » State is argument to every routine
- Means long and complicate control structure
  - » Case statements everywhere on state
- Violates single choice principle
  - » Too many locations need to know about all states
    - > difficult to modify as states added or removed
- **Not reusable/general – except as a template**
  - » **implicit argument in all functions is the application**
  - » **Generality → know about all states in all applications**

OO Solution?

How do you use OO to solve the problem?

An OO Solution

Routines exchange too much data ?

→ Put routines in your data

An OO Solution – 2

Routines exchange too much data ?
→ put routines in your data

- **Instead of building components around operations while distributing data**
  - » **OO does reverse**
    - > **build around data and distribute operations**

An OO Solution – 3

Routines exchange too much data ?
→ put routines in your data

- Instead of building components around operations while distributing data
  - » OO does reverse
    - > build around data and distribute operations
- **Use most important data types as basis for modules**
  - » **Routines are attached to data to which it relates most closely**

An OO Solution – 4

Routines exchange too much data ?
→ put routines in your data

- Instead of building components around operations while distributing data
  - » OO does reverse
    - > build around data and distribute operations
- Use most important data types as basis for modules
  - » Routines are attached to data to which it relates most closely
- **In this application state should be a class**

State as Class

- What would be handed over to state?

State as Class – 2

- What would be handed over to state?
  - » **All operations that characterize a state**
    - > **???**

State as Class – 3

- What would be handed over to state?
  - » **All operations that characterize a state**
    - > **Displaying screen**
    - > **Analyzing answer**
    - > **Checking answer**
    - > **Producing error messages**
    - > **Processing correct answer**
  - » **Customize for each state**

Class State

STATE*

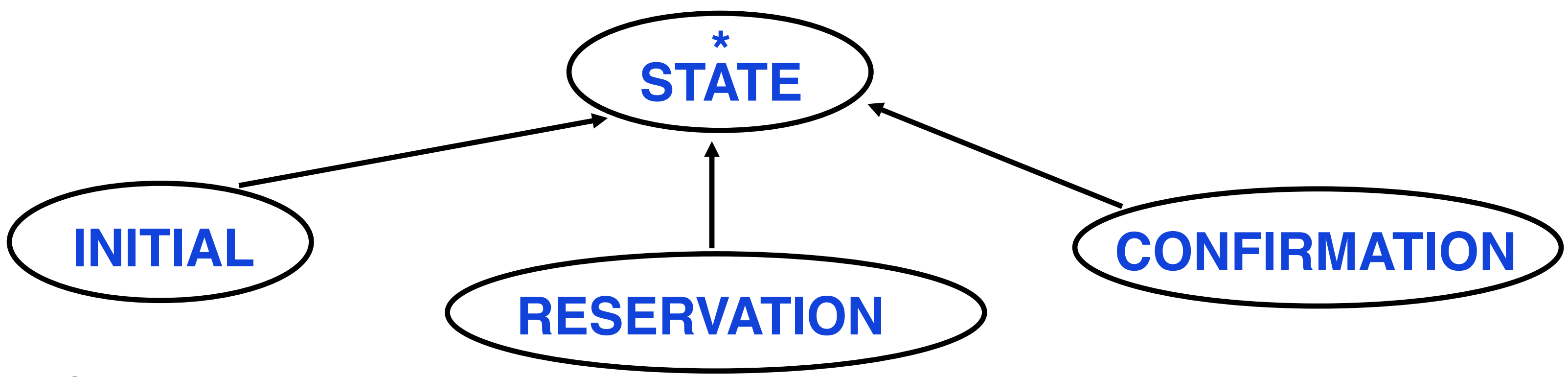
input : ANSWER
choice : INTEGER
execute
correct : BOOLEAN
display*
read*
message*
process*

- Deferred class
- Deferred features
- Execute is effective because we know its behaviour

```
execute
local ok : BOOLEAN
do
   from not ok until ok loop
   display  ;  read  ;  ok := correct
   if not ok then message end
   end
ensure ok
end
```

Inheritance & Implementation

- **STATE** describes the general notion of state
  - » **execute is the same for all states**
  - » **other routines must be customized**
- Use deferred classes to specify general situation and provide for extension
- Use inheritance to specify particular states
  - » **Implement deferred routines**



Architecture of System

- Separates elements common to all states and elements specific to individual states

- Common elements do not need to be redeclared in descendants

Architecture of System – 2

- Satisfies **open-closed principle**
  - » **STATE is closed**
  - » **Inheritance opens it**
- State is typical of **behaviour classes**
  - » **Deferred classes capture common behaviour**
- Inheritance & Deferral are key for reusable components

Completing the System Design

- How do we represent transitions and an actual application?

- Have to take care of managing a session
  - » **What execute_session did in top down**

- What is missing?
  - » **The notion of the specific application**

Application Class

- Features
  - » **execute**
    - > **how to execute the application**
  - » **initial & is_final**
    - > **special states – properties of application**
  - » **transition**
    - > **mapping from state to state**
- May want to add more features
  - » **Add new state or transition**
  - » **Store in a data base**
  - » **...**

Application Class – 2

```
class application feature
   initial_state_number : INTEGER

   execute
      local state : STATE  ;  state_number : INTEGER
      do
      from st_number : initial_state_number
      until st_number = 0
      loop
         state := associated_state.item ( state_number )
         state.execute
         state_number := transition.item (state.number, state.choice )
      end
          …
          More detail in next 4 slides
          …
end
```

Transition Array

- Number states from 1..P
  Choices are numbered from 1..Q

- Represent transition as an P (states) x Q(choices) array **transition**

```
transition : ARRAY2 [ INTEGER ]  -- State numbers
```

- Need support routines such as the following

```
put_transistion ( in_state_number : INTEGER
                ; choice : INTEGER
                ; out_state_number : INTEGER )
```

Associated State

» Array **associated_state** gives the STATE associated with a state number

```
associated_state : ARRAY [ STATE ]
```

- Need support routines such as the following

```
put_state ( state_number : INTEGER
          ; state : STATE )
```

Initial state

- Attribute **initial_state_number** represents the starting state
    - **Have a support routine to select the initial state**

set_initial_state (state_number : INTEGER)

Implementing the Design

- Creation procedure of **APPLICATION** uses creation procedures of ARRAY and ARRAY2
  - **see p691 & 692 of Meyer 1997**

- Building an application is relatively easy due separation of parts

Points to Think About

- Forget about a main program
- Focus on data abstraction
  - » **Leads to structures that can more easily change and are more easily reused**
- Don't ask
  - » **What does the system do?**
    - > **It is not a function**

Points to Think About – 2

- Don't worry too much about modelling the **real world**
  - » **Goto version is a close model but poor design**

Points to Think About – 3

- Heuristic to find the classes
  - » **Look for data transmissions and concepts that appear in communication between numerous components of a system**

Points to Think About – 4

What counts in OO design is how good are your abstractions for structuring your software.

Points to Think About – 5

What counts in OO design is how good are your abstractions for structuring your software.

Above all else, worry about finding the right abstractions

Points to Think About – 6

What counts in OO design is how good are your abstractions for structuring your software.

Above all else, worry about finding the right abstractions

Big win from OO

clear, general, manageable, change-ready abstractions