ETH Zürich

Chair of Software Engineering

Software Architecture

Bertrand Meyer, Carlo A. Furia, Martin Nordio

ETH Zurich, February-May 2011

Lecture 1: Introduction

Goal of the course

(From the course description in the ETH page)

Software Architecture covers two closely related aspects of software technology:

- Techniques of software design: devising proper modular structures for software systems. This is "architecture" in the strict sense.
- An introduction to the non-programming, non-design aspects of software engineering.

Some topics

Software architecture:

- Modularity and reusability
- Abstract Data Types
- Design by Contract and other O-O principles
- Design Patterns
- Component-Based Development
- Designing for concurrency

Software engineering:

- Process models
- Requirements analysis
- CMMI and agile methods
- Cost estimation
- Software metrics
- Software testing
- Configuration management
- Project management

Plus: an introduction to UML

Practical information

Lecturers

Bertrand Meyer, bertrand.meyer@inf.ethz.ch
Office: RZ J22

Carlo A. Furia, carlo.furia@inf.ethz.ch
Office: RZ J4

Martin Nordio, martin.nordio@inf.ethz.ch
Office: RZ J3

Assistants

Julian Tschannen (head assistant)

Christian Estler

(Max) Yu Pei

Marco Piccioni

(Jason) Yi Wei

Course material

Course page:

http://se.inf.ethz.ch/teaching/2011-F/Soft_Arch-0050/

→ Check it regularly

Lecture material:

- Lecture slides
- Recommended textbooks:

  B. Meyer: *Object-Oriented Software Construction, 2nd edition* -- Prentice Hall, 1997

  E. Gamma et al.: *Design Patterns*
  Addison-Wesley, 1995

Exercise material:

- Exercise sheets
- Master solutions

Supplementary recommended books

A good software engineering textbook (see precise references on course page):

- Ghezzi / Jazayeri / Mandrioli
  (broadest scope)
- Pfleeger / Atlee
  (the most recent)
- Pressman
  (emphasis on practitioners' needs)

On patterns: Karine Arnout's ETH PhD thesis (available electronically)

Electronic forums

Discussion forums:
Hosted by Inforum (VIS):
http://forum.vis.ethz.ch

Make sure you are registered online in "MyStudies"

To email the whole teaching team (professor and assistants):

se-softarch-assi@lists.inf.ethz.ch

Grading

50% project, 50% end-of-semester exam

To pass the course, you need a 4.0 (at least) in both the project and the exam.

About the exam:

- When: Tuesday, 31 May 2011, 13-15 (normal class time), 90 minutes
- What: all topics of semester
- How: no material allowed ("closed-book")

About the project

The project is an integral part of the course

Goal:

- Apply software architecture techniques
- Practice group work in software engineering
- Go through main phases of a realistic software project: requirements, design of both program and test plan, implementation, testing

Project groups

The project must be done in groups of 4 students (smaller groups are allowed only in special circumstances).

You must form the groups soon (by Friday 25 -- this week!)

Once you have a group, send one email per group to Julian Tschannen (julian.tschannen@inf.ethz.ch) with the names of the group members and their Origo usernames

- register on origo.ethz.ch if you have no account yet

If you can't find a group, send us an email with your name and Origo username, so we can put you together with other students.

Project topic

This year's topic is to develop:

- An application programming interface (API) for relational database access

(you will use Eiffel for both design and implementation)

Project deadlines*

*May be subject to slight adaptation

1. Requirements specification:
   - Handed out: 28 February
   - Due: 20 March
2. API design:
   - Handed out: 21 March
   - Due: 10 April
3. Implementation:
   - Handed out: 11 April
   - Due: 8 May
4. Testing:
   - Handed out: 9 May
   - Due: 29 May

More details on the project

Grading criteria for each step, and the weight for each step, are given on the Web page

We will use SVN on Origo for source control. All submissions (documents and source code) will be delivered through this repository. You will have to create an Origo project for your team. See the Web page for details.

Standards

For each step (except implementation), you will be given a template and will have to follow it

While the project involves programming, it is not primarily a programming project, but a software engineering project. You will discover some of the challenges and techniques of developing software as part of actual projects.

On forming the groups:

- Select partners with complementary skills, e.g. requirements, documentation, design, programming

A request

We do not want you to drop the course, but if you are going to do so, please drop out early (March 10 at the latest) out of courtesy to other students

What is software architecture?

Software architecture

We define software architecture as

*The decomposition of software systems into modules**

Primary criteria: extendibility and reusability

Examples of software architecture techniques & principles:

- Abstract data types (as the underlying theory)
- Object-oriented techniques: the notion of class, inheritance, dynamic binding
- Object-oriented principles: uniform access, single-choice, open-closed principle…
- Design patterns
- Classification of software architecture styles, e.g. pipes and filters

* From the title of an article by Parnas, 1972

Software architecture: milestones

1968: *The inner and outer syntax of a programming language* (Maurice Wilkes)

1968-1972: Structured programming (Edsger Dijkstra); industrial applications (Harlan Mills & others)

1971: *Program Development by Stepwise Refinement* (Niklaus Wirth)

1972: David Parnas's articles on information hiding

1974: Liskov and Zilles's paper on abstract data types

1975: *Programming-in-the-large vs Programming-in-the-small* (Frank DeRemer & Hans Kron)

1987: *Object-Oriented Software Construction*, 1st edition

1994: *An introduction to Software Architecture* (David Garlan and Mary Shaw)

1995: *Design Patterns* (Erich Gamma et al.)

1997: UML 1.0

What is software engineering?

A definition of software engineering

Wikipedia (from SWEBOK, the Software Engineering Body of Knowledge)

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software.

(Largely useless definition.)

A simpler definition

"The application of engineering to software"

Engineering (Wikipedia): "the discipline, art and profession of acquiring and applying technical, scientific, and mathematical knowledge to design and implement materials, structures, machines, devices, systems, and processes that safely realize a desired objective or invention"

A simpler definition of engineering: the application of scientific principles to the construction of artifacts

Parnas's view

(Cited in Ghezzi et al.)

"The multi-person construction of multiversion software"

For this course

The application of engineering principles and techniques, based on mathematics, to the development and operation of possibly large software systems satisfying defined standards of quality

"Large" software systems

What may be large: any or all of

- Source size (lines of code, LoC)
- Binary size
- Number of users
- Number of developers
- Life of the project (decades...)
- Number of changes, of versions

(Remember Parnas's definition)

Process and product

Software engineering affects both:

- Software products
- The processes used to obtain and operate them

Products are not limited to code. Other examples include requirements, design, documentation, test plans, test results, bug reports

Processes exists whether they are formalized or not

Software quality factors

Product

Immediate

- Correctness
- Robustness
- Security
- Ease of use
- Ease of learning
- Efficiency

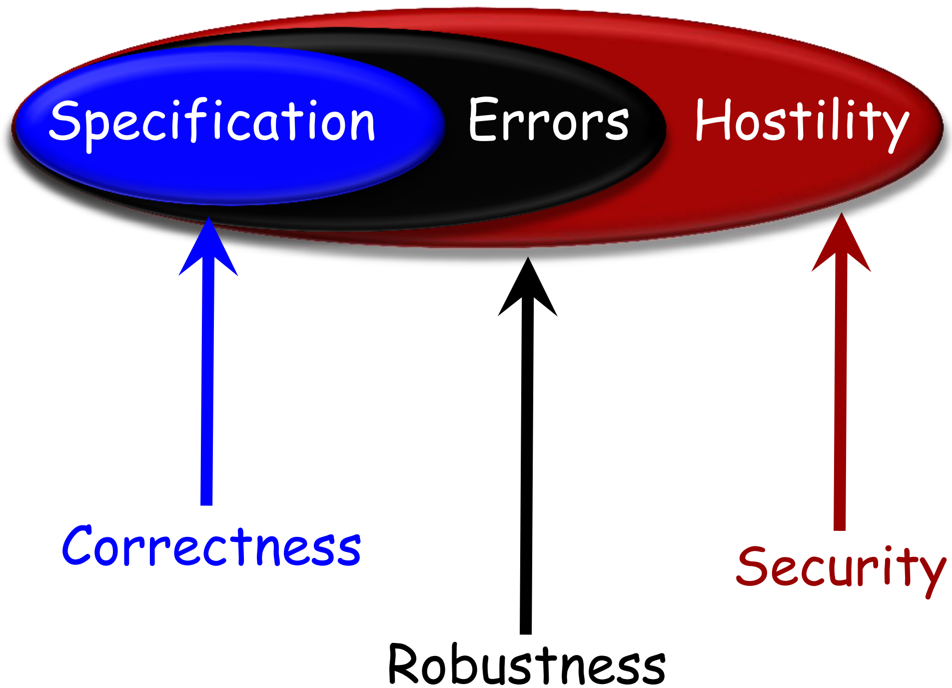
Long-term

- Extendibility
- Reusability
- Portability

Process

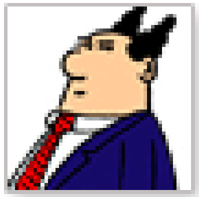
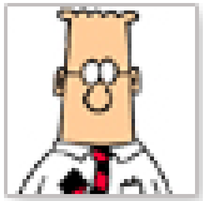
- Timeliness
- Cost-effectiveness

"Reliability"



Software engineering today

Three cultures:

- Process 
- Agile 
- Object

The first two are usually seen as exclusive, but all have major contributions to make.

Process

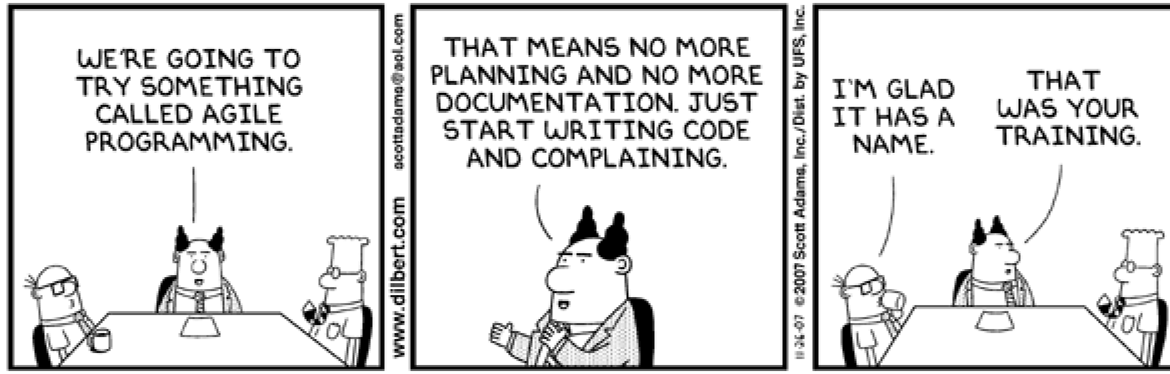
Emphasize:

- Plans
- Schedules
- Documents
- Requirements
- Specifications
- Order of tasks
- Commitments

Examples: Rational Unified Process, CMMI, Waterfall…

Agile

Emphasize:



- Short iterations
- Emphasis on working code; de-emphasis of plans and documents
- Emphasis on testing; de-emphasis of specifications and design . "Test-Driven Development"
- Constant customer involvement
- Refusal to commit to both functionality and deadlines
- Specific practices, e.g. Pair Programming

Examples: Extreme Programming (XP), Scrum

Object-oriented culture

Emphasizes:

- Seamless development
- Reversibility
- Single Product Principle
- Design by Contract

Six task groups of software engineering

- **Describe** — Requirements, design specification, documentation …
- **Implement** — Design, programming
- **Assess** — V&V*, esp. testing
- **Manage** — Plans, schedules, communication, reviews…
- **Operate** — Deployment, installation,
- **Notate** — Languages for programming etc.

** Validation & Verification*

A software architecture example

Our first pattern example

Multi-panel interactive systems

Plan of the rest of this lecture:

- Description of the problem: an example
- An unstructured solution
- A top-down, functional solution
- An object-oriented solution yielding a useful design pattern
- Analysis of the solution and its benefits

A reservation panel

Flight sought from: Santa Barbara To: Zurich

Depart no earlier than: 18 Feb 2010 No later than: 19 Feb 2010

ERROR: Choose a date in the future

Choose next action:

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

A reservation panel

Flight sought from: Santa Barbara To: Zurich

Depart no earlier than: 18 Feb 2011 No later than: 19 Feb 2011

AVAILABLE FLIGHTS: 2

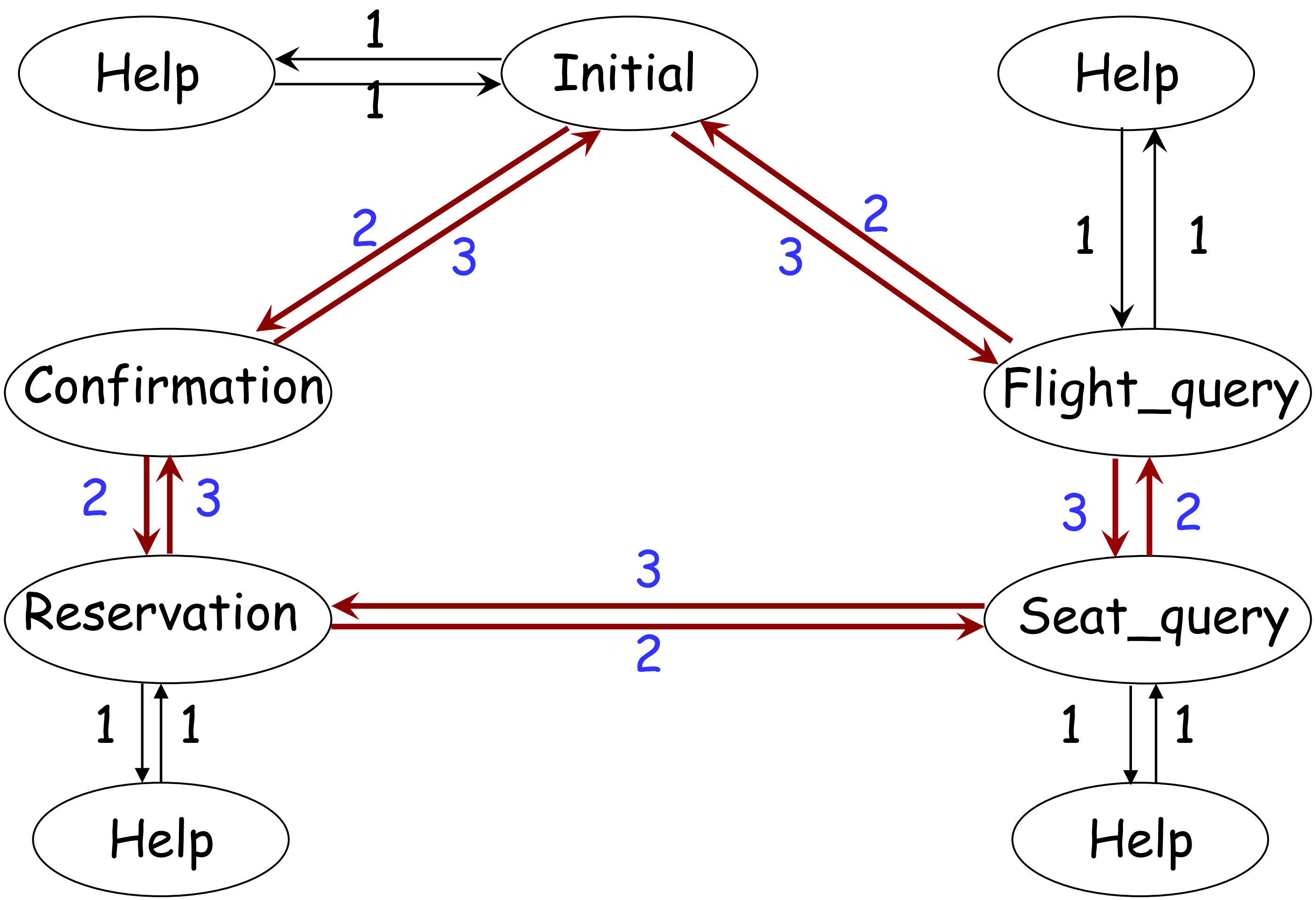
Flt# UA 425 Dep 8:25 Arr 7:45 Thru: LAX, JFK

Flt# AA 082 Dep 7:40 Arr 9:15 Thru: LAX, DFW

Choose next action:

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

The transition diagram



A first attempt

A program block for each state, for example:

P_{Flight_query}:

```
display "enquiry on flights" screen
repeat
    Read user's answers and his exit choice C
    if Error_in_answer then output_message end
until
    not Error_in_answer
end

process answer

inspect C
    when 0 then goto P_Exit
    when 1 then goto P_Help
    ...
    when n then goto P_Reservation
end
```

What's wrong with the previous scheme?

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

A functional, top-down solution

Represent the structure of the diagram by a function

transition (*i*, *k*)

giving the state to go to from state *i* for choice *k*.

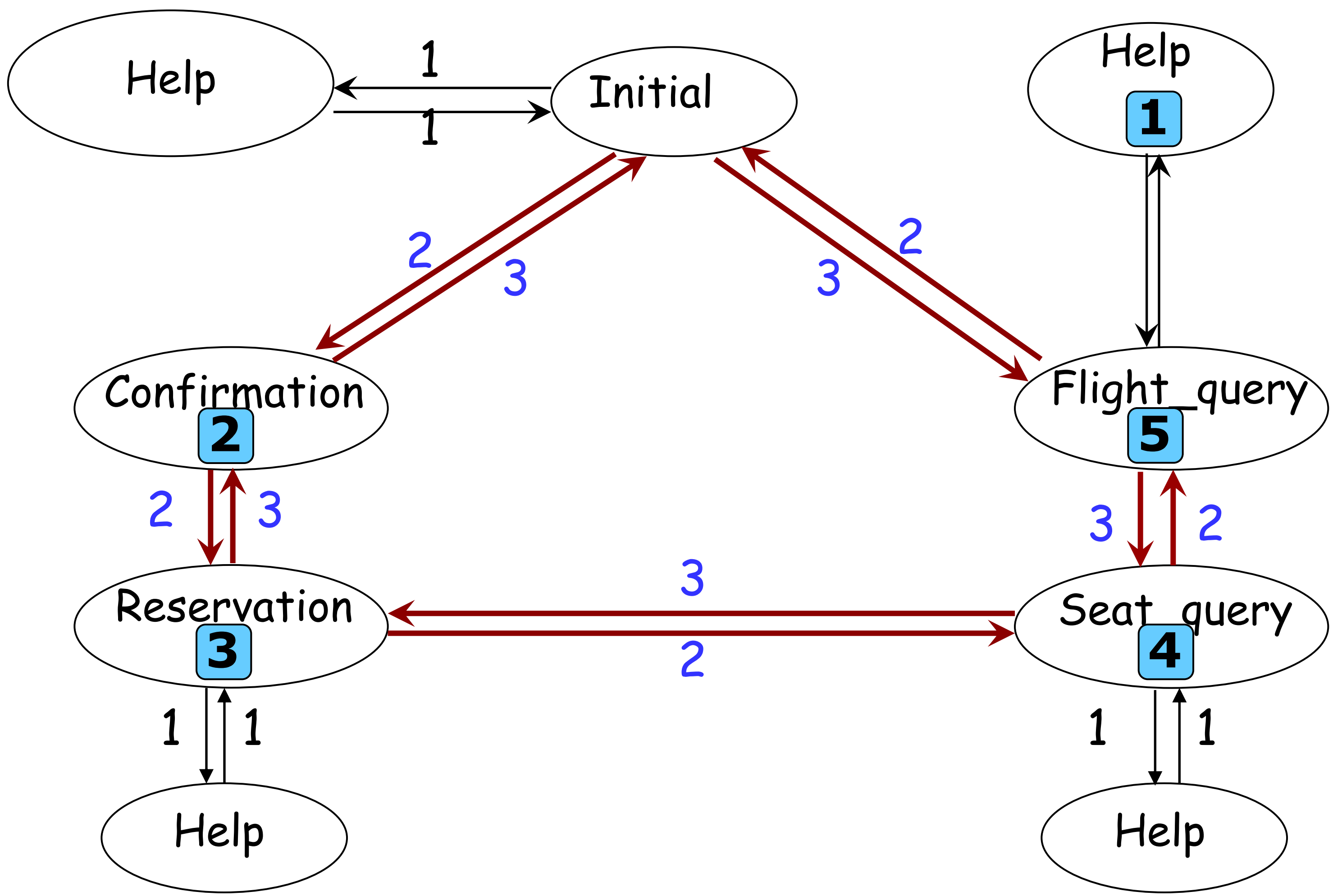
This describes the transitions of any particular application.

Function *transition* may be implemented as a data structure, for example a two-dimensional array.

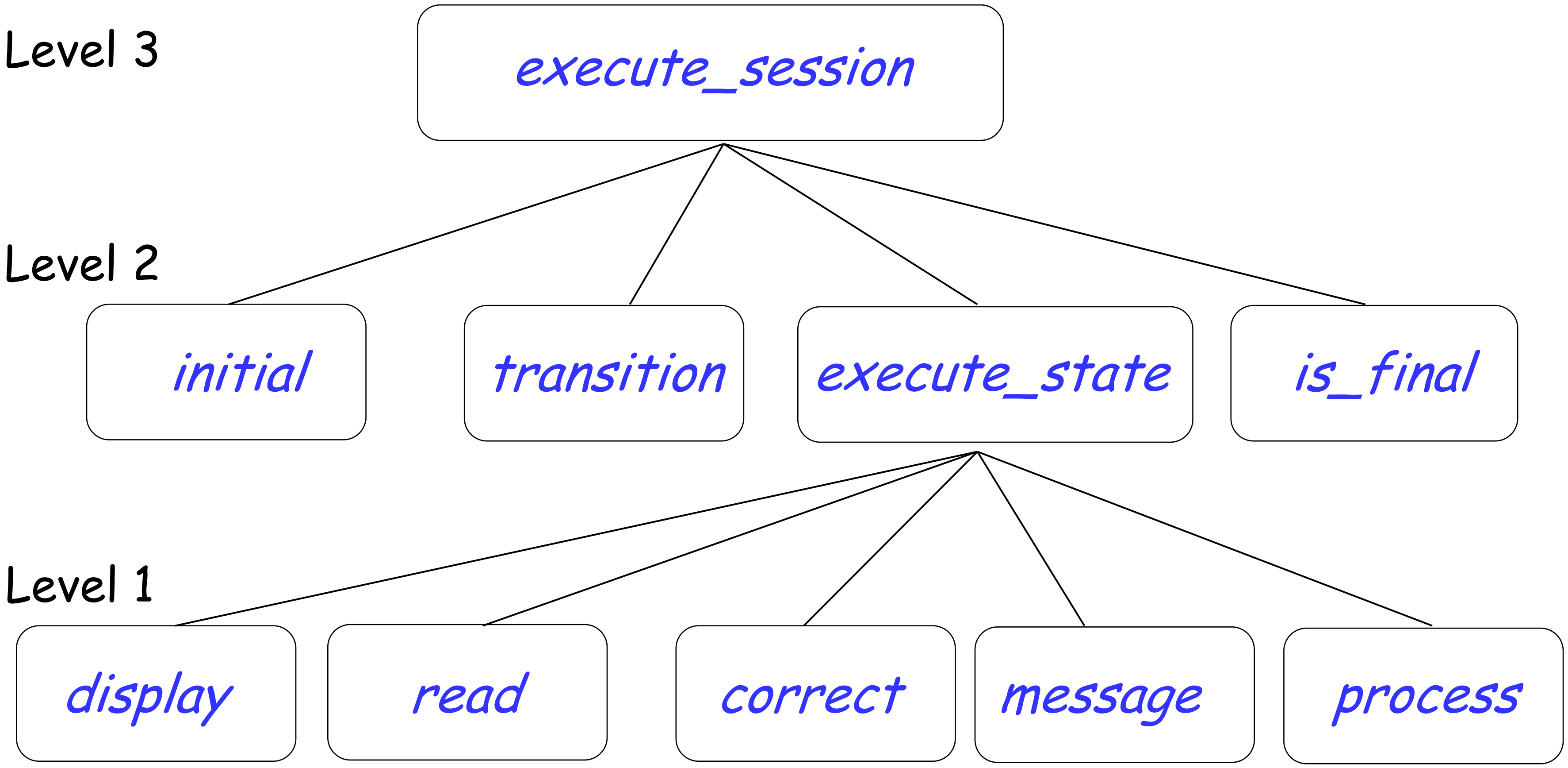
The transition function

	0	1	2	3
0 (Initial)			2	
1 (Help)	*Exit*	*Return*		
2 (Confirmation)	*Exit*		3	0
3 (Reservation)	*Exit*		4	2
4 (Seats)	*Exit*		5	3
5 (Flights)	*Exit*		0	4

The transition diagram



New system architecture



New system architecture

Procedure *execute_session* only defines graph traversal.

It knows nothing about particular screens of a given application; it should be the same for all applications.

```
execute_session
        -- 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
```

To describe an application

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

Actions in a state

```
execute_state (current_state : INTEGER ): INTEGER
        -- Execute actions for current_state ; return 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 )
        Result := choice
end
```

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.

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

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

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

The flow of control

Underlying reason why structure is so inflexible:

Too much DATA TRANSMISSION.

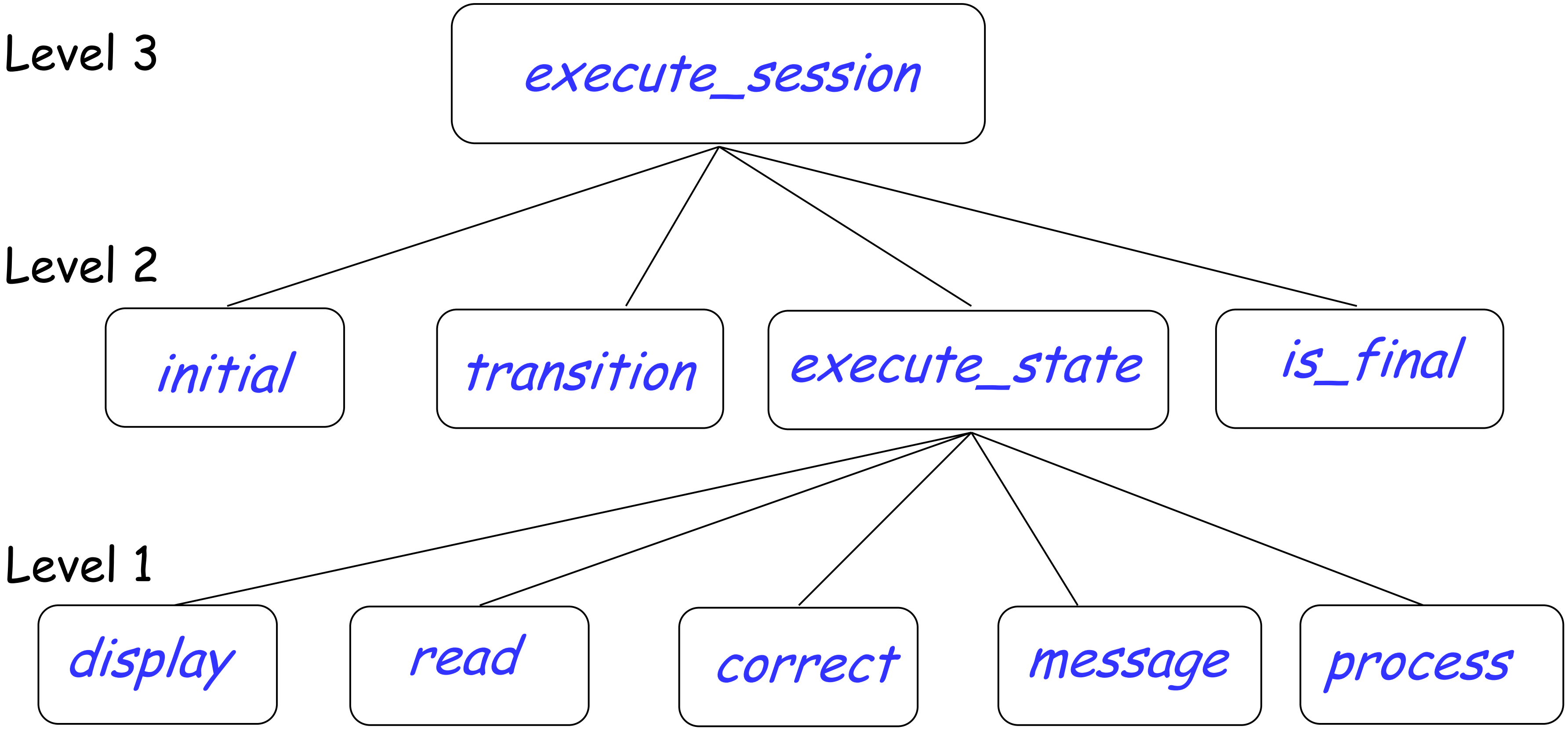
current_state is passed from *execute_session* (level 3) to all routines on level 2 and on to level 1

Worse: there's another implicit argument to all routines – application. Can't define

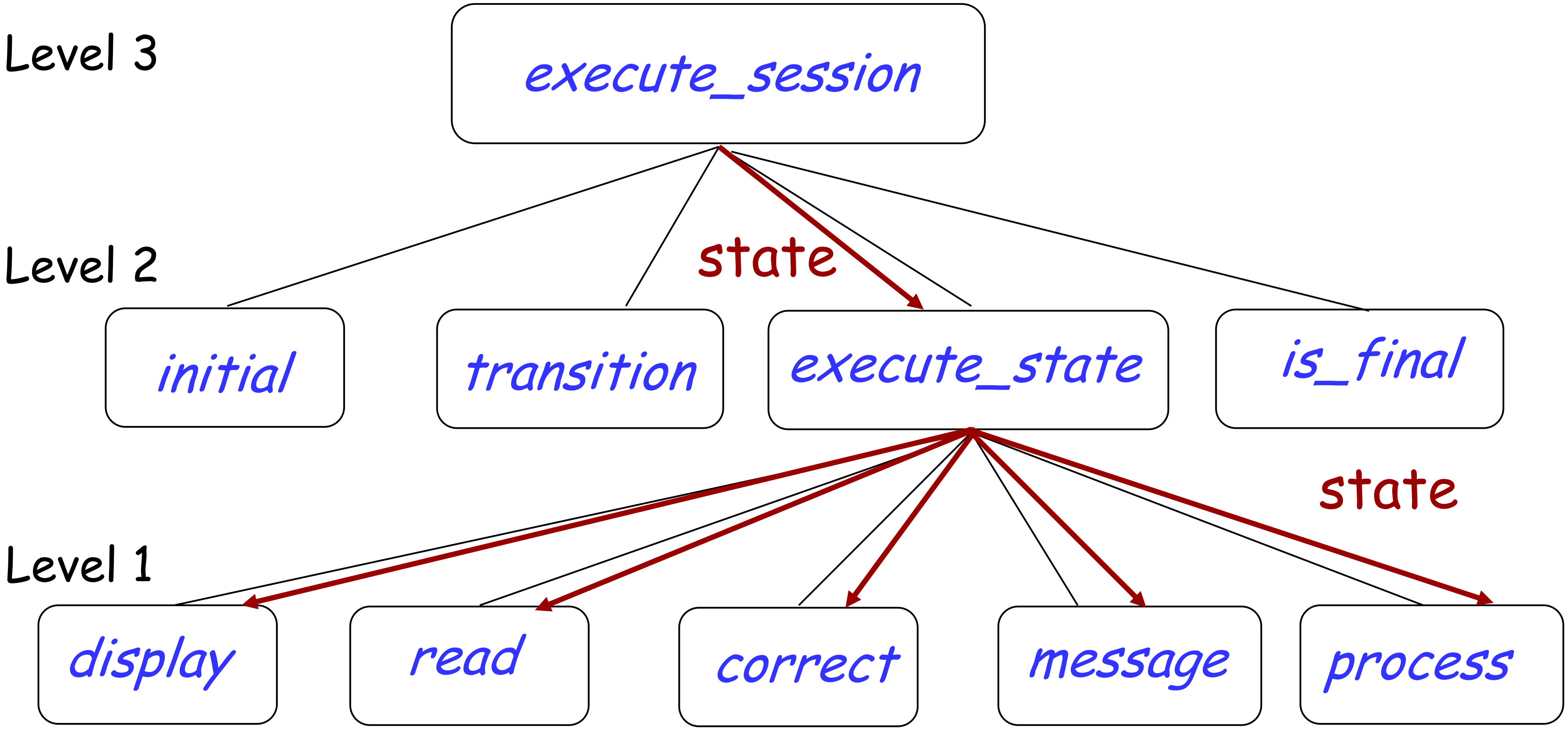
execute_session, *display*, *execute_state*, ...

as library components, since each must know about all interactive applications that may use it.

The visible architecture



The real story



The law of inversion

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

In this example: the state is everywhere!

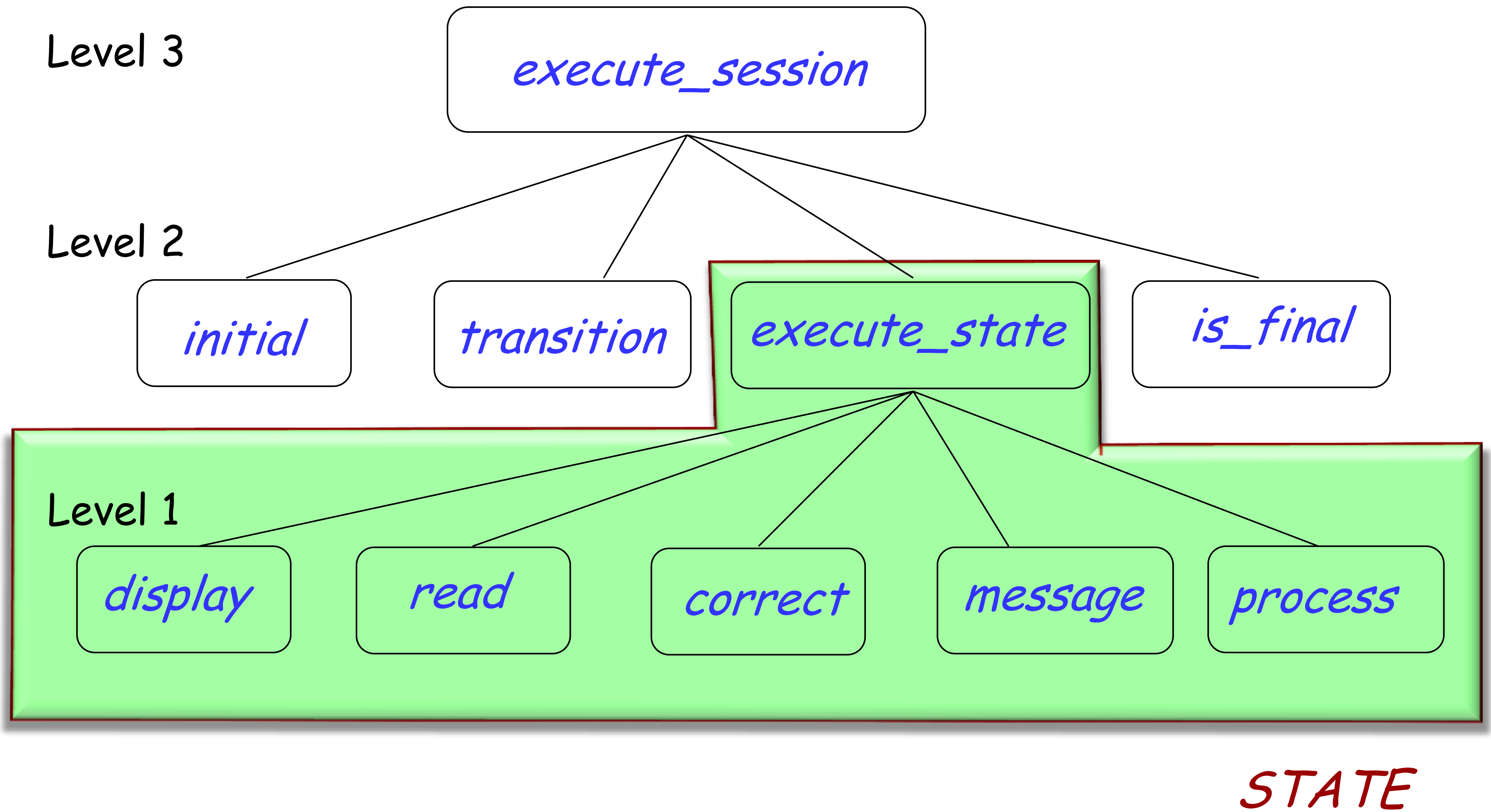
Going O-O

Use *STATE* as the basic abstract data type (and class).

Among features of every state:

- The routines of level 1 (deferred in class *STATE*)
- *execute_state*, as above but without the argument *current_state*

Grouping by data abstractions



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

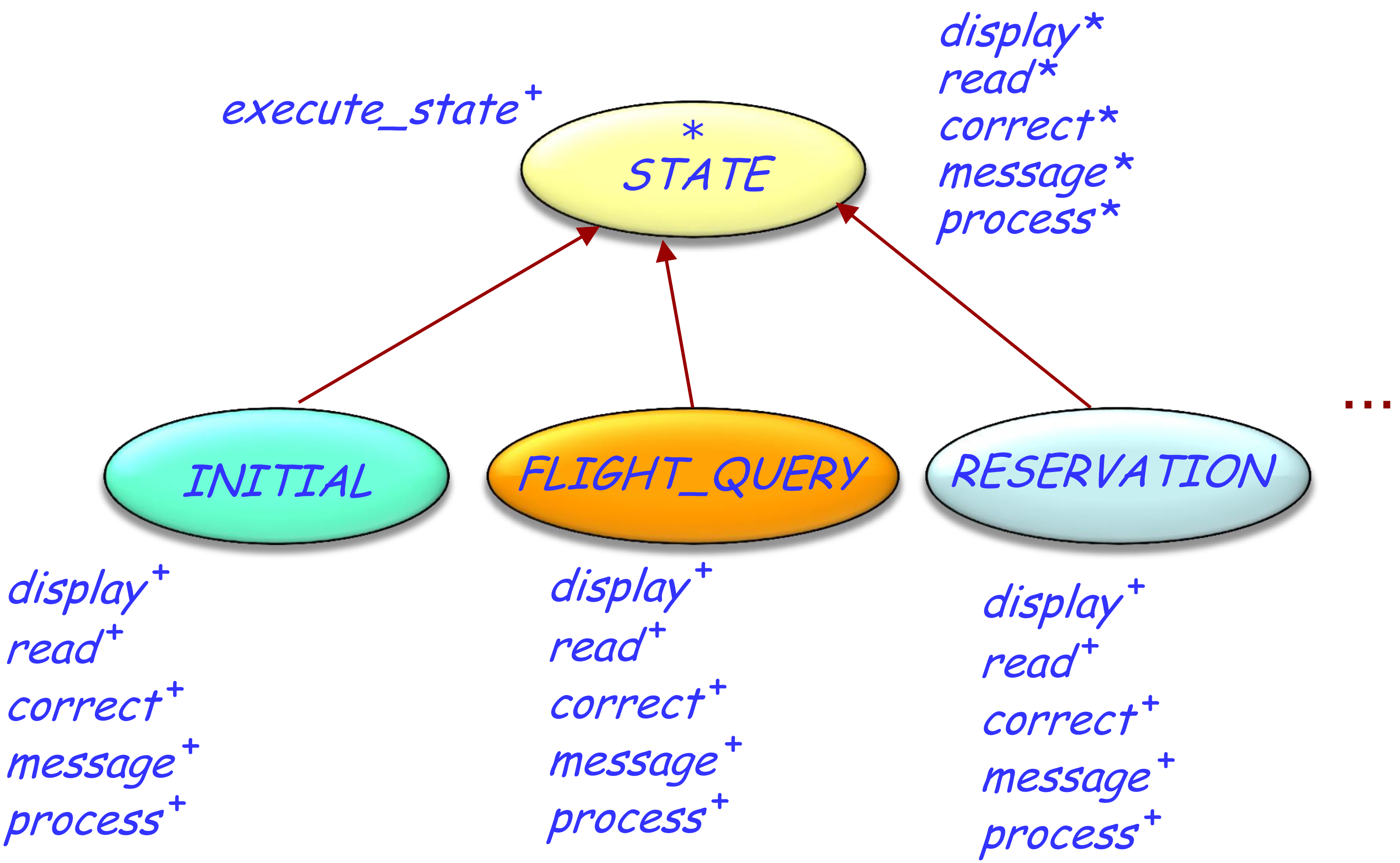
Class *STATE*

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

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

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

Class *STATE*

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

Class structure

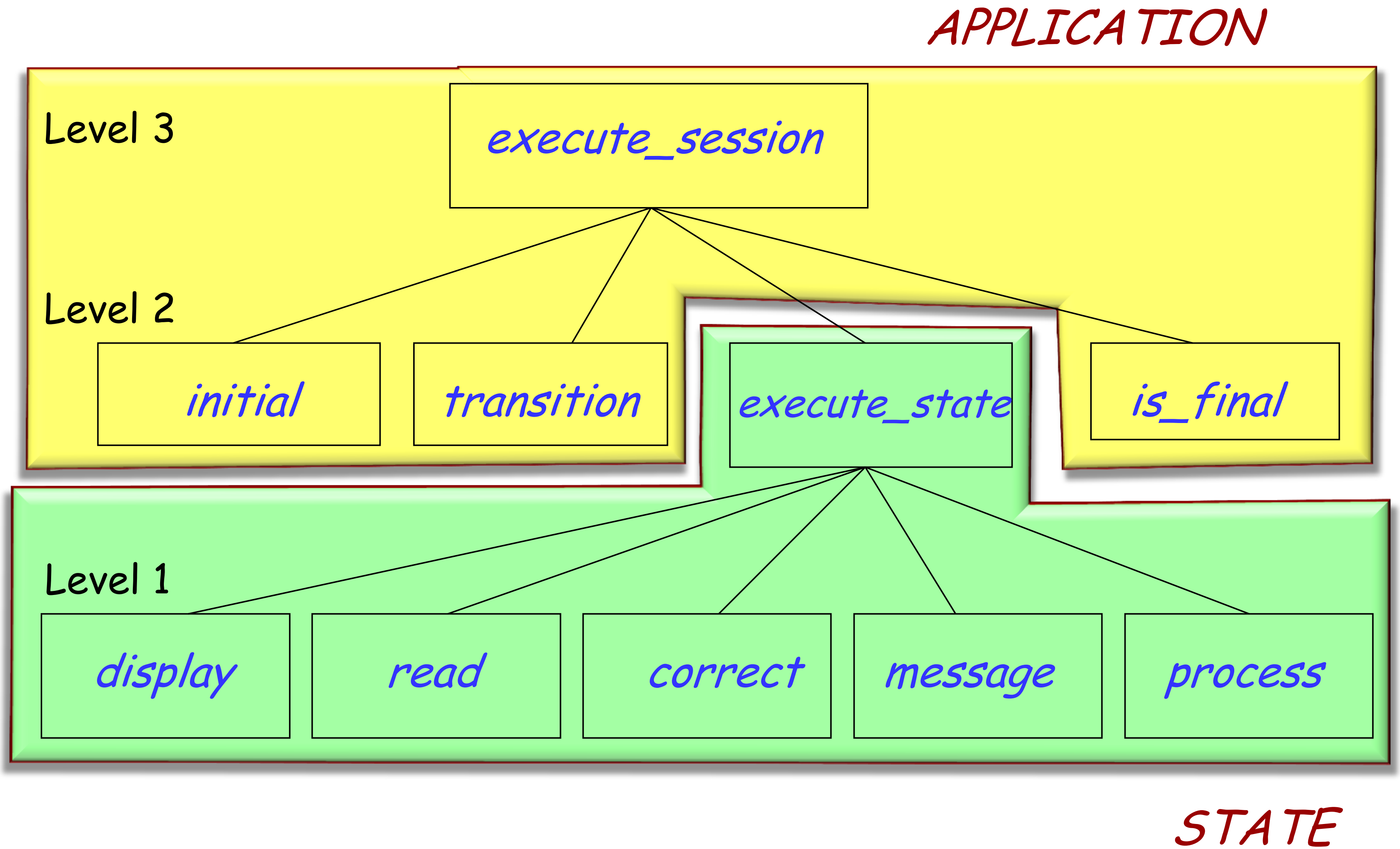


To describe a state of an application

Write a descendant of *STATE*:

```
class FLIGHT_QUERY inherit
        STATE
feature
        display do ... end

        read do ... end

        correct : BOOLEAN do ... end

        message do ... end

        process do ... end
end
```

Rearranging the modules



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?)

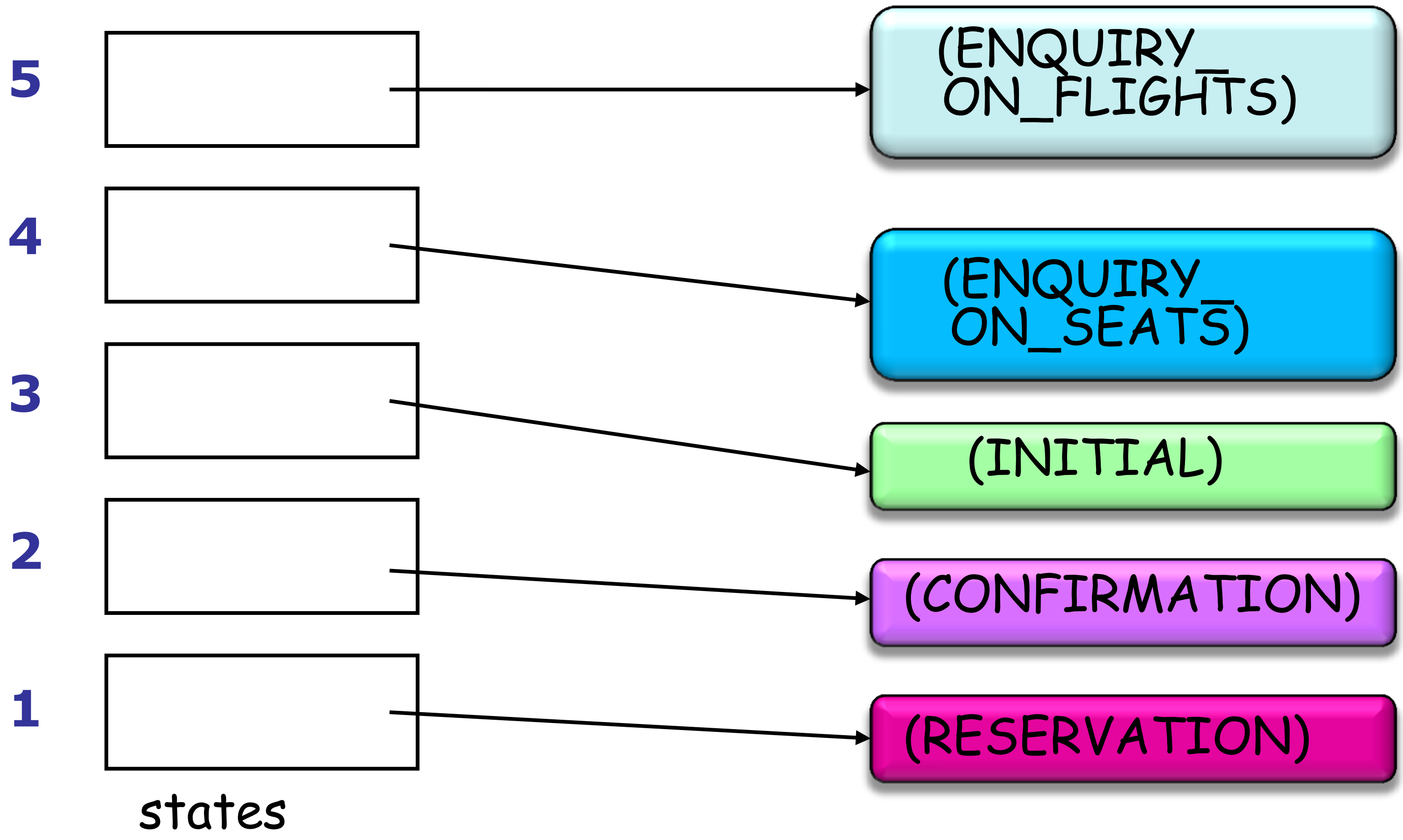
- No deferred boolean function *is_final*, but convention: a transition to state 0 denotes termination.

- No such convention for initial state (too constraining). Attribute *initial_number*.

Describing an application

```
class
        APPLICATION
create
        make

feature

        initial : INTEGER

        make (n, m : INTEGER)
                -- 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
```

The array of states

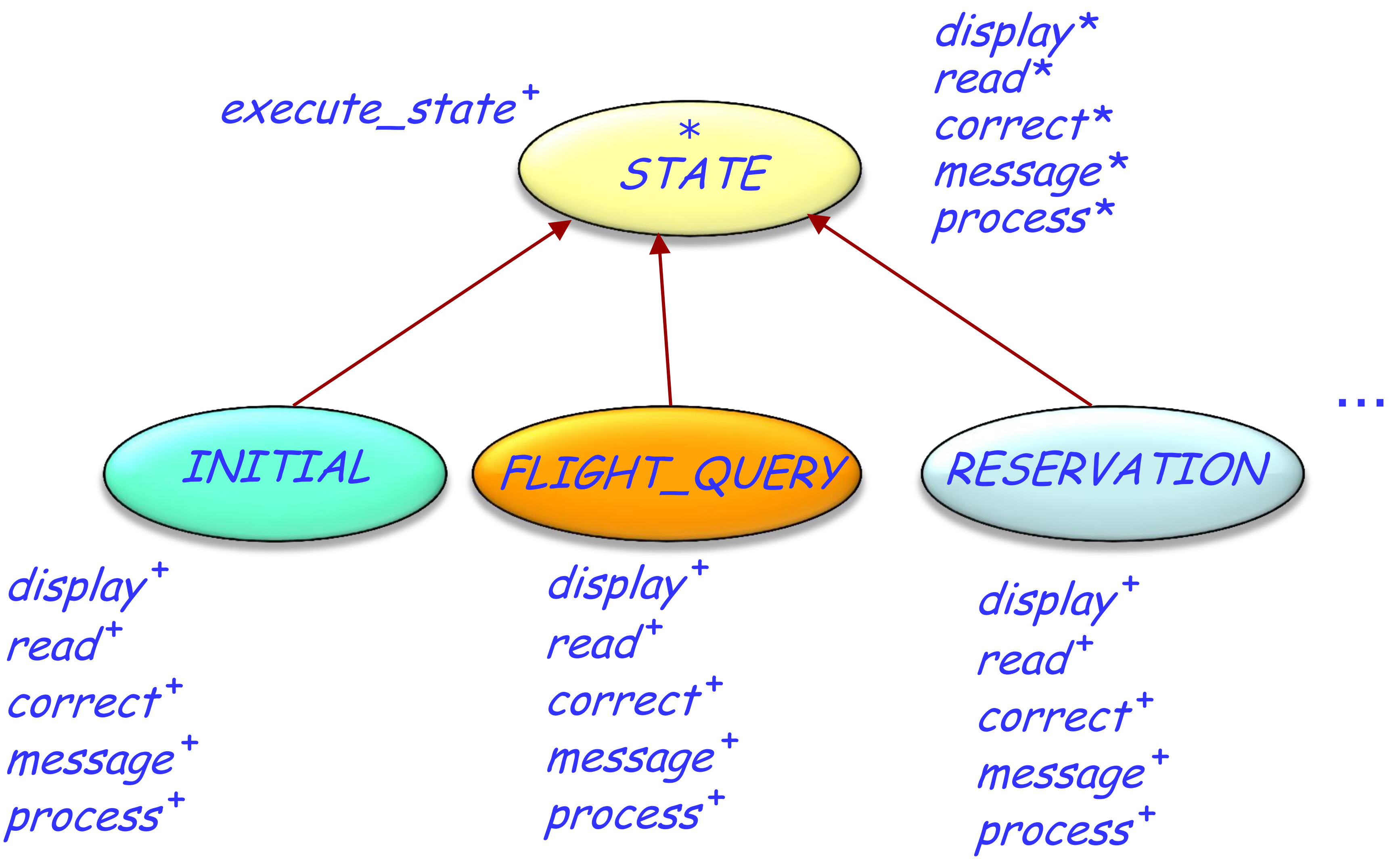


A polymorphic data structure!

Executing a session

```
execute_session
            -- Run one session of application
    local
            current_state : STATE          -- Polymorphic!
            index : INTEGER
    do
            from
                    index := initial
            until
                    index = 0
            loop
                    current_state := states [index ]
                    current_state.execute_state
                    index := transition [index, current_state.choice]
            end
    end
```

Class structure

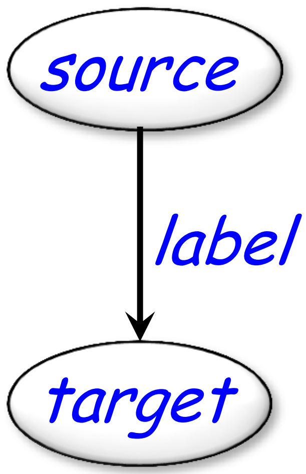


Other features of *APPLICATION*

```
put_state (s: STATE; number: INTEGER)
            -- Enter state s with index number
    require
            1 <= number
            number <= states.upper
    do
            states [number] := s
    end

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

More features of *APPLICATION*

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



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 [*n*] := *s*

Choose initial state *n0*:

a.choose_initial (*n0*)

Enter transitions:

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

May now run:

a.execute_session

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 design pattern

"State and Application"

Software architecture: the basic issue

Finding the right data abstractions

What we have seen

Basic definitions and concepts of software engineering

Basic definitions and concepts of software architecture

A design pattern: State and Application

The role of data abstraction

Techniques for finding good data abstractions