Software Architecture

Bertrand Meyer

ETH Zurich, March-July 2007

Lecture 1: Introduction

Goal of the course

Introduce you to the techniques of building large software systems of high quality, in particular:
- Reliability
- Extendibility
- Reusability

This includes in particular:
- Principles of software quality
- Object technology principles and methods; the practice of object-oriented analysis, design and implementation
- Design patterns
- Principles of building reusable software
- Some key techniques of concurrent programming

Six key topics

- Modularity and reusability
- Abstract Data Types
- Design by Contract and other O-O principles
- Design Patterns
- Component-Based Development
- Introduction to concurrency
Practical information

Course material

Course page:
http://se.inf.ethz.ch/teaching/ss2007/0050/
Check it at least twice a week

Lecture material:
- Lecture slides
- Textbook:
  Object-Oriented Software Construction,
  Available from Polybuchhandlung (~ CHF 63 with Legi)

Exercise material:
- Exercise sheets
- Master solutions

Electronic forums

Discussion forums:
Inforum:
http://forum.vis.ethz.ch

Mailing list for each group

Usual advice and rules:
- Use the forums and mailing lists! Take advantage of every help you can get.
- Don't be shy. There are no stupid questions.
- Criticism welcome, but always be polite to every participant and observe the etiquette.
- To email the whole teaching team (professor and assistants):
  soft-arch-assi@se.inf.ethz.ch
**Exercise sessions and project**

Make sure to attend all sessions

Exercise sheets will be distributed by your assistant during the exercise session

Do all exercises and the project

---

**Start of semester**

No exercise session this week

Next week: single-group exercise session led by Bernd Schoeller; room will be announced

Exercise groups will be formed next week

---

**Project**

Details to be given early April

You will have the choice between four topic categories:

- TRAFFIC extension or improvement
- Games using EiffelMedia
- Open project to be discussed with assistant
- EiffelStudio extension or improvement

All projects will be done in Eiffel

EiffelStudio download:


Open-source version available for Windows, Linux and MacOS
This is a software architecture project

- Design quality is essential
- Group project, must be managed properly
- Configuration management
- Documentation
- Quality standards (analysis, design, implementation)
- Should be useful ("Eat your own dog food!")

The public presentation

- All projects will be demonstrated
- The best projects will be selected for presentation

Exam: end of semester

- Tuesday, 19 June 2007, 14-16 (normal class time)
- 2-hour exam
- No material allowed
- Covers all material in the semester
Teaching staff

Bertrand Meyer

E-mail: Bertrand.Meyer@inf.ethz.ch
Office: RZ J6
Secretary: Claudia Günthart, (01) 632 83 46

Exercise sessions

All groups have one session a week:
- Thursday, 15:00-16:00
The assistants

- Martin Nordio (Coordinating Assistant)
  English
- Ilinca Ciupa
  English
- Michela Pedroni
  German
- Bernd Schoeller
  German
- Till Bay
  German (French)
- Jason (Yi) Wei
  English

End lecture 1

Software Architecture

Bertrand Meyer

ETH Zurich, March-July 2007

Lecture 2: A basic 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 Mar 2006] No later than: [18 Mar 2006]
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 Mar 2006] No later than: [18 Mar 2006]

AVAILABLE FLIGHTS: 2
Flt LH 425 Dep 8:25 Arr 7:45 Thru: Shanghai
Flt CP 082 Dep 7:40 Arr 9:15 Thru: Hong Kong

Choose next action:
0 – Exit
1 – Help
2 – Further enquiry
3 – Reserve a seat
A first attempt

A program block for each state, for example:

```
PFlight_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 PExit
    when 1 then goto PHelp
    ... when n then goto PReservation
```

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

\[ \text{transition}(i, k) \]

giving the state to go to from state \( i \) for choice \( k \).

This describes the transitions of any particular application.

Function \( \text{transition} \) may be implemented as a data structure, for example a two-dimensional array.

---

The transition function

<table>
<thead>
<tr>
<th></th>
<th>0 (Initial)</th>
<th>1 (Help)</th>
<th>2 (Confirmation)</th>
<th>3 (Reservation)</th>
<th>4 (Seats)</th>
<th>5 (Flights)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Initial)</td>
<td>Exit</td>
<td></td>
<td>Exit</td>
<td></td>
<td>Exit</td>
<td>Exit</td>
</tr>
<tr>
<td>1 (Help)</td>
<td>Return</td>
<td>Exit</td>
<td>Exit</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2 (Confirmation)</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3 (Reservation)</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
</tr>
<tr>
<td>4 (Seats)</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
</tr>
<tr>
<td>5 (Flights)</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
</tr>
</tbody>
</table>

---

The transition diagram

![Diagram showing transitions](image-url)
New system architecture

Procedure \texttt{execute\_session} only defines graph traversal. It knows nothing about particular screens of a given application; it should be the same for all applications.

\begin{verbatim}
execute_session is
  local -- Execute full session
do
  current_state, choice : INTEGER
  current_state := initial
  repeat
    choice := execute_state(current_state)
    current_state := transition(current_state, choice)
  until is_final(current_state)
end
\end{verbatim}

To describe an application

- Provide \texttt{transition} function
- Define \texttt{initial} state
- Define \texttt{is\_final} function
**Actions in a state**

```plaintext
execute_state (current_state : INTEGER): INTEGER is
    -- 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

    return 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) is
do
  inspect current_state
  when state1 then
  when state2 then
  when state_n 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.

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

The 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 \textit{STATE}

\begin{verbatim}
  deferred class STATE
  feature
    choice: INTEGER  -- User's selection for next step
    input: ANSWER    -- User's answer for this step
    display is
      deferred
    end
    read is
      deferred
    end
    ensure
      input /= Void
    end
  ensure
    correct: BOOLEAN  -- Is input acceptable?
  ensure
    message: -- Display message for erroneous input
      require not correct
      deferred
    end
  ensure
    process is
      require correct
      deferred
    end
\end{verbatim}
Class **STATE**

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

Class structure

![Class structure diagram](image)

To describe a state of an application

```plaintext
To describe a state of an application

Write a descendant of **STATE**:  

```
Rearranging the modules

APPLICATION

Level 3
execute_session

Level 2
initial transition execute_state is_final

STATE
Level 1
display read correct message process

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

```plaintext
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
```

The array of states

```
STATES
  5 (ENQUIRY
     ON_FLIGHTS)
  4 (ENQUIRY
     ON_SEATS)
  3 (INITIAL)
  2 (CONFIRMATION)
  1 (RESERVATION)
```

A polymorphic data structure!

Executing a session

```plaintext
execute_session is -- 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

```
STATE

- execute_state
- display
- read
- correct
- message
- process

INITIAL

- display
- read
- correct
- message
- process

FLIGHT_QUERY

- display
- read
- correct
- message
- process

RESERVATION

- display
- read
- correct
- message
- process

...```

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
```

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
```
To build an application

Necessary states — instances of \texttt{STATE} — should be available.

Initialize application:
\begin{itemize}
\item create \texttt{a.make} (state\_count, choice\_count)
\end{itemize}

Assign a number to every relevant state \textit{s}:
\begin{itemize}
\item \texttt{a.put\_state} (s, n)
\end{itemize}

Choose initial state \textit{n0}:
\begin{itemize}
\item \texttt{a.choose\_initial} (n0)
\end{itemize}

Enter transitions:
\begin{itemize}
\item \texttt{a.put\_transition} (sou, tar, lab)
\end{itemize}

May now run:
\begin{itemize}
\item \texttt{a.execute\_session}
\end{itemize}

Open architecture

During system evolution you may at any time:
\begin{itemize}
\item Add a new transition (\texttt{put\_transition}).
\item Add a new state (\texttt{put\_state}).
\item Delete a state (not shown, but easy to add).
\item Change the actions performed in a given state
\item ...
\end{itemize}

Note on the architecture

Procedure \texttt{execute\_session} is not "the function of the system" but just one routine of \texttt{APPLICATION}.

Other uses of an application:
\begin{itemize}
\item Build and modify: add or delete state, transition, etc.
\item Simulate, e.g. in batch (replaying a previous session's script), or on a line-oriented terminal.
\item Collect statistics, a log, a script of an execution.
\item Store into a file or data base, and retrieve.
\end{itemize}

Each such extension only requires incremental addition of routines. Doesn't affect structure of \texttt{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

Object-Oriented Design

It's all about finding the right data abstractions