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 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 list! 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: http://www.eiffel.com/downloads/
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
FM LH 425 Dep 8:25 Arr 7:45 Thru: Shanghai
FM 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:

\[ P_{\text{Flight query}} \]

- 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 \( P_{\text{Exit}} \)
- when 1 then goto \( P_{\text{Help}} \)
- when n then goto \( P_{\text{Reservation}} \)

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

The transition diagram

New system architecture

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

\[
\text{execute_session} \quad \text{is} \quad \text{-- Execute full session}
\]

\[
\text{local} \quad \text{current_state, choice : INTEGER}
\]

\[
\text{do}
\]

\[
\text{current_state} := \text{initial}
\]

\[
\text{repeat}
\]

\[
\text{choice} := \text{execute_state} (\text{current_state})
\]

\[
\text{current_state} := \text{transition} (\text{current_state}, \text{choice})
\]

\[
\text{until} \quad \text{is_final} (\text{current_state})
\]

\[
\text{end}
\]

New system architecture

To describe an application

- Provide \( \text{transition} \) function
- Define \( \text{initial} \) state
- Define \( \text{is_final} \) function
Actions in a state

```
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) end
    until good
  end
process (current_state, answer)
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 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`

def class STATE

def feature

`choice`: INTEGER -- User's selection for next step
`input`: ANSWER -- User's answer for this step
`display`: -- Show screen for this step.
  deferred end
`read`: -- Get user's answer and exit choice, -- recording them into input and choice.
  deferred ensure input /= Void
  end

ensure

Class `STATE`

def class STATE

def feature

`correct`: BOOLEAN is -- 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`

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

Class `FLIGHT_QUERY` inherit `STATE`

```java
feature
display is do ... end
read is do ... end
correct : BOOLEAN is do ... end
message is do ... end
process is do ... end
end
```

To describe a state of an application

Write a descendant of `STATE`:

```java
class FLIGHT_QUERY 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
```

Rearranging the modules

```
Level 3
execute_state

Level 2
initial transition execute_state
is_final

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

```
The array of states
STATES
1
2
3
4
5
(ENQUIRY_ON_FLIGHTS)
(ENQUIRY_ON_SEATS)
(INITIAL)
(CONFIRMATION)
(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
    until
    loop
        current_state := states [index]
        current_state.execute_state
        index := transition [index, current_state.choice]
end
```

Class structure

```
Class structure
execute_state
* STATE
- INITIAL
- FLIGHT_QUERY
- RESERVATION
- ...
- display*
- read*
- correct*
- message*
- process*
```

Other features of APPLICATION

```plaintext
put_state (s : STATE; number : INTEGER) is
    -- Enter state s with index number.
    require
    do
        number <= states.upper
        states.put (number, s)
end
choose_initial (number : INTEGER) is
    -- Define state number number as the initial state.
    require
    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
    do
        source <= states.upper
        target <= states.upper
        label <= transition.upper2
        transition.put (source, label, target)
end
```
To build an application

Necessary states — instances of STATE — should be available.

Initialize application:
- create $a.make\left(\text{state\_count}, \text{choice\_count}\right)$

Assign a number to every relevant state $s$:
- $a.put\_state\left(s, n\right)$

Choose initial state $n_0$:
- $a.choose\_initial\left(n_0\right)$

Enter transitions:
- $a.put\_transition\left(s_{ou}, t_{ar}, l_{ab}\right)$

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

Object-Oriented Design

It's all about finding the right data abstractions