Software Architecture

Bertrand Meyer, Michela Pedroni

ETH Zurich, February-May 2010

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
- 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

Michela Pedroni,  Michela.Pedroni@inf.ethz.ch
Office: RZ J8
Assistants

Dr. Martin Nordio
Stefan van Staden
Julian Tschannen
(Jason) Yi Wei
Course material

Course page:
http://se.inf.ethz.ch/teaching/2010-S/0050/
→ Check it regularly

Lecture material:
- Lecture slides
- Recommended textbooks:
  B. Meyer: *Object-Oriented Software Construction*,

  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:
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):
  se-soft-arch-assi@lists.inf.ethz.ch
Grading

50% project, 50% \textit{end-of-semester} exam

\textbf{About the exam:}

- \textbf{When:} Tuesday, 1 June 2010, 14-16 (normal class time), 90 minutes
- \textbf{What:} all topics of semester
- \textbf{How:} no material allowed
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

The project may be done in groups: 1 to 4 students

You must form the groups soon (by Friday of this week)

If you are looking for partners and need help please use our “dating service”
Project topic

Choice between two topics:

- A data structure visualization library (for use e.g. in a debugger)
- An HTML generation library
1. Requirements specification:
   - Handed out: 1 March
   - Due: 21 March, Presentation: 22 March

2. API design:
   - Due: 11 April, Presentation: 12 April

3. Test plan (for another team's project):
   - Due: 25 April, Presentation: 26 April

4. Implementation (in Eiffel)
   - Due: 16 May, Presentation: 17 May

5. Test report (on other team's implementation):
   - Due: 23 May

6. Final release
   - Due: 4 June

*Subject to slight adaptation*
Notes on the project

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

In step 3 (previous slide) will be asked to devise a test plan for another group’s project, based only on its requirements specification (prior to implementation). Then you will have to run the test.
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.

Forming the groups:

- You do not need to reach the maximum team size (4). Think of the tradeoffs involved: more manpower or more communication problems?
- 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 drop out early (March 25 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
- 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
Characterizing software architecture

“The inner and outer syntax of a programming language” (Wilkes, 1968)

“Programming-in-the-large” vs “Programming-in-the-small” (DeRemer & Kron, 1975)
What is software engineering?
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: 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
Process and product

Software engineering affects both:

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

Products are not limited to code. Other examples include requirements, designs, documentation

Processes exist 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

Specification

Errors

Hostility

Correctness

Robustness

Security
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
- Testing (over specifications); “Test-Driven Development"
- Constant customer involvement
- Refusal to commit to both functionality and deadlines
- Specific practices, e.g. Pair Programming

Examples: Extreme Programming (XP), lean programming
Object-oriented culture

Emphasizes:

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

*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 2009  No later than: 18 Feb 2009

ERROR: Choose a date in the future

Choose next action:
0  – Exit
1  – Help
2  – Further enquiry
3  – Reserve a seat
Flight sought from: Santa Barbara  To: Zurich
Depart no earlier than: 18 Mar 2010  No later than: 18 Mar 2010

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
The transition diagram

Help → Initial → Help

Initial → Confirmation → Flight_query

Initial → Reservation → Seat_query

Reservation → Help

Flight_query → Help

Seat_query → Help

Help → Confirmation

Help → Reservation

Help → Flight_query

Help → Seat_query

Help → Help
A first attempt

A program block for each state, for example:

\[ P_{\text{Flight\_query}}: \]

\[
\text{display "enquiry on flights" screen}
\]
\[
\text{repeat}
\quad \text{Read user's answers and his exit choice} \ C
\quad \text{if Error\_in\_answer then output\_message end}
\]
\[
\text{until}
\quad \text{not Error\_in\_answer}
\]
\[
\text{end}
\]

\text{process answer}

\[
\text{inspect} \ C
\quad \text{when 0 then goto} \ P_{\text{Exit}}
\quad \text{when 1 then goto} \ P_{\text{Help}}
\quad \text{...}
\quad \text{when} \ n \ \text{then goto} \ P_{\text{Reservation}}
\]
\[
\text{end}
\]
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

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

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

This describes the transitions of any particular application.

Function \textit{transition} may be implemented as a data structure, for example a two-dimensional array.
The transition function

<table>
<thead>
<tr>
<th>State</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>Return</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
</tr>
<tr>
<td>1 (Help)</td>
<td>Exit</td>
<td>Return</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
</tr>
<tr>
<td>2 (Confirmation)</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</td>
<td>Exit</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
New system architecture

Level 3
- `execute_session`

Level 2
- `initial`
- `transition`
- `execute_state`
- `is_final`

Level 1
- `display`
- `read`
- `correct`
- `message`
- `process`
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.

```plaintext
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

\[ \text{execute\_state}(\text{current\_state} : \text{INTEGER} ) : \text{INTEGER} \]

-- Execute actions for current\_state; return user's exit choice.

local

answer : \text{ANSWER} 

good : \text{BOOLEAN} 

choice : \text{INTEGER} 

do 

repeat 

\[ \text{display}(\text{current\_state}) \]

\[ [\text{answer}, \text{choice}] := \text{read}(\text{current\_state}) \]

\[ \text{good} := \text{correct}(\text{current\_state}, \text{answer}) \]

\[ \text{if not good then} \quad \text{message}(\text{current\_state}, \text{answer}) \quad \text{end} \]

until 

\[ \text{good} \]

end

\[ \text{process}(\text{current\_state}, \text{answer}) \]

Result := \text{choice} 

end
Specification of the remaining routines

- $\text{display}(s)$ outputs the screen associated with state $s$.

- $[a, e] := \text{read}(s)$ reads into $a$ the user’s answer to the display screen of state $s$, and into $e$ the user’s exit choice.

- $\text{correct}(s, a)$ returns true if and only if $a$ is a correct answer for the question asked in state $s$.

- If so, $\text{process}(s, a)$ processes answer $a$.

- If not, $\text{message}(s, a)$ 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)`
All routines share the state as input argument. They must discriminate on it, e.g.:

```plaintext
display (current_state: INTEGER)
  do
    inspect current_state
    when state\(_1\) then
    \hspace{1cm} \ldots
    when state\(_2\) then
    \hspace{1cm} \ldots
    when state\(_n\) then
    \hspace{1cm} \ldots
  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

Level 3
- execute_session

Level 2
- initial
- transition
- execute_state
- is_final

Level 1
- display
- read
- correct
- message
- process
The real story

Level 3
- execute_session

Level 2
- initial
- transition
- execute_state
- is_final

Level 1
- display
- read
- correct
- message
- process
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 \textit{STATE} as the basic \texttt{abstract data type} (and class).

Among features of every state:

- The routines of level 1 (deferred in class \textit{STATE})
- \texttt{execute\_state}, as above but without the argument \texttt{current\_state}
Grouping by data abstractions

Level 3
- `execute_session`

Level 2
- `initial`
- `transition`
- `execute_state`
- `is_final`

Level 1
- `display`
- `read`
- `correct`
- `message`
- `process`
Class **STATE**

defered class

    **STATE**

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
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*

```plaintext
execute_state
  local
    good: BOOLEAN
  do
    from
    until
      good
      loop
        display
        read
        good := correct
        if not good then message end
      end
    end
  process
    choice := input.choice
end
end
```
To describe a state of an application

Write a descendant of \textit{STATE}:

\begin{verbatim}
class FLIGHT_QUERY inherit STATE

feature
  display do ... end
  read do ... end
  correct : BOOLEAN do ... end
  message do ... end
  process do ... end
end
\end{verbatim}
Rearranging the modules

APPLICATION

Level 3
execute_session

Level 2
initial
transition
execute_state
is_final

Level 1
display
read
correct
message
process

STATE
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

- **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

\texttt{put\_state (s: STATE; number: INTEGER)}

\begin{center}
\begin{tabular}{c}
\hline
\texttt{-- Enter state } s \texttt{ with index number}
\hline
\texttt{require}\n\hline
\begin{tabular}{c}
1 \leq number
\end{tabular}
\hline
\texttt{do}\n\hline
\begin{tabular}{c}
number \leq states.upper
\end{tabular}
\hline
\texttt{end}\n\hline
\texttt{states.put (number, s)}
\hline
\end{tabular}
\end{center}

\texttt{choose\_initial (number: INTEGER)}

\begin{center}
\begin{tabular}{c}
\hline
\texttt{-- Define state number } number \texttt{ as the initial state.}
\hline
\texttt{require}\n\hline
\begin{tabular}{c}
1 \leq number
\end{tabular}
\hline
\begin{tabular}{c}
number \leq states.upper
\end{tabular}
\hline
\texttt{do}\n\hline
\texttt{first\_number := number}
\hline
\texttt{end}\n\hline
\end{tabular}
\end{center}
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:

```c
create a.make (state_count, choice_count)
```

Assign a number to every relevant state `s`:

```c
a.put_state (s, n)
```

Choose initial state `n0`:

```c
a.choose_initial (n0)
```

Enter transitions:

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

May now run:

```c
a.execute_session
```
Open architecture

During system evolution you may at any time:

- Add a new transition \((\text{put\_transition})\).
- Add a new state \((\text{put\_state})\).
- 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