Software Architecture

Bertrand Meyer
ETH Zurich, 30 March – 29 June 2005

Lecture 2: Software quality; principles and criteria of modularity
Agenda for today

- Software quality
- Modularity
- Reusability
OOSC, chapters
  3: Modularity
  6: Abstract data types
The collection of processes, methods, techniques, tools and languages for developing quality operational software.
“Software” is not just programs. It includes processes for producing and operating the programs, as well as all related documentation,
Software engineering

The collection of processes, methods, techniques, tools and languages for developing quality operational software.
Software quality: external vs internal

- **External** factors: visible to customers
  
  (not just end users but e.g. purchasers)
  
  - *Examples*: ease of use, extendibility, timeliness

- **Internal** factors: perceptible only to developers
  
  - *Examples*: good programming style, information hiding

Only external factors count in the end, but the internal factors make it possible to obtain them.
Software quality: product vs process

- **Product**: properties of the resulting software

  For example: correctness, efficiency

- **Process**: properties of the procedures used to produce and “maintain” the software
External quality factors

Product quality (immediate):
- Correctness
- Robustness
- Security
- Ease of use
- Ease of learning
- Efficiency

Product quality (long-term):
- Extendibility
- Reusability
- Portability

Process quality:
- Timeliness
- Cost-effectiveness
Reliability

Correctness:
The systems’ ability to perform according to specification, in cases covered by the specification

Robustness:
The systems’ ability to perform reasonably in cases not covered by the specification

Security (integrity):
The systems’ ability to protect itself against hostile use
Security

Carcassonne, 13th century
Chambord, 16th century
Versailles, 17-18th century
“I think that it’s extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every now and then, and after a while we began to take their complaints seriously. We began to feel as if we really were responsible for the successful, error-free perfect use of these machines. I don’t think we are. I think we’re responsible for stretching them, setting them off in new directions, and keeping fun in the house.”

Alan Perlis, Preface to *The Structure and Interpretation of Computer programs* by Abelson and Sussman
External quality factors

- CORRECTNESS
- ROBUSTNESS
- INTEGRITY

- EASE OF USE
- REUSABILITY
- EXTENDIBILITY
- PORTABILITY
- EFFICIENCY
- ...

- Correctness:
  - The ability of a software system to perform according to specification, in cases defined by the specification.

- Robustness:
  - The ability of a software system to react in a reasonable manner to cases not covered by the specification.
Reliability

- Correctness + Robustness + Security

- Techniques will be studied in detail: typing, Design by Contract, ...
Modularity

- Reusability + Extendibility

- Favored by architectural techniques tending to ensure decentralization of modules
Modularity

Some principles of modularity:
- Decomposability
- Composability
- Continuity
- Information hiding
- The open-closed principle
- The single choice principle
Decomposability

- Method helps decompose complex problems into subproblems.

- COROLLARY: Division of labor.
  - Example: Top-down design method (see next).
  - Counter-example: General initialization module.
Top-down functional design

Topmost functional abstraction

A
  /\    /\        /\  /\   /\   /\
B  D  C
  /\  | |\  | |\  | |  | |  | |  | |
C1  I  C2  I1  I2

Sequence

Loop

Conditional
Top-down design


http://www.acm.org/classics/dec95/
Composability

- Method favors production of software elements that may be freely combined with each other to produce new software.

- Example: Unix shell conventions
  Program1 | Program2 | Program3
Direct mapping

- Method yields software systems whose modular structure remains compatible with any modular structure devised in the process of modeling the problem domain.
Few interfaces principle

- Every module communicates with as few others as possible.
Small interfaces principle

- If two modules communicate, they exchange as little information as possible.
Explicit interfaces principle

- Whenever two modules $A$ and $B$ communicate, this is obvious from the text of $A$ or $B$ or both.

Diagram:
- Module A
- Module B
- Data item $x$
  - Module A modifies $x$
  - Module B accesses $x$
- Method ensures that small changes in specifications yield small changes in architecture.

- Design method: Specification $\rightarrow$ Architecture

- Example: Principle of Uniform Access (see next)

- Counter-example: Programs with patterns after the physical implementation of data structures.
Uniform Access Principle

- Facilities managed by a module are accessible to its clients in the same way whether implemented by computation or by storage.

- Definition: A client of a module is any module that uses its facilities.
Uniform Access: An example

\[
\text{balance} = \text{list_of_deposits.total} - \text{list_of_withdrawals.total}
\]

\[
\begin{align*}
\text{list_of_deposits} & \quad \text{list_of_withdrawals} \\
\text{balance} & \\
\end{align*}
\]

(A1)

(A2)

Ada, Pascal, C/C++, Java, C#:  
\text{a.balance} \\
\text{balance(a)} \\

Simula, Eiffel:  
\text{a.balance} \\
\text{a.balance}()}
Information hiding

- Underlying question: how does one “advertise” the capabilities of a module?

- Every module should be known to the outside world through an official, “public” interface.

- The rest of the module’s properties comprises its “secrets”.

- It should be impossible to access the secrets from the outside.
The designer of every module must select a subset of the module’s properties as the official information about the module, to be made available to authors of client modules.
Information hiding

Public part

Secret part
Information hiding

- Justifications:
  - Continuity
  - Decomposability
The Open-Closed Principle

- Modules should be open and closed.

- Definitions:
  - Open module: May be extended.
  - Closed module: Usable by clients. May be approved, baselined and (if program unit) compiled.

- The rationales are complementary:
  - For closing a module (manager’s perspective): Clients need it now.
  - For keeping modules open (developer’s perspective): One frequently overlooks aspects of the problem.
An object has an interface
An object has an implementation
Information hiding
The Open-Closed principle
Closing modules prematurely

\[
\text{type } \textit{PUBLICATION} =
\begin{array}{l}
\text{record} \\
\quad \text{author, title: STRING;}
\quad \text{publication_year: INTEGER}
\qquad \text{case } \text{subtype: (book, journal, conference) of}
\qquad \text{book: (publisher: STRING);}
\qquad \text{journal: (editor: STRING);}
\qquad \text{conference: (place, chair: STRING)}
\end{array}
\end{array}
\]

\[\text{end end}\]

- Use in clients:

\[
p: \textit{PUBLICATION} ;
\text{case } p.\text{subtype } \text{of}
\quad \text{book: ... } p.\text{publisher } ... ;
\quad \text{journal: ... } p.\text{editor } ... ;
\quad \text{conference: ... } p.\text{place } ... 
\text{end}
\]
The Single Choice principle

- Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.
  - Editor: set of commands (insert, delete etc.)
  - Graphics system: set of figure types (rectangle, circle etc.)
  - Compiler: set of language constructs (instruction, loop, expression etc.)
Reusability issues

- **Organizational and managerial issues:**
  - (Not covered here.)

- **Technical issues: what form of components?**
  - Routine libraries
  - Packages (Ada)
  - Class libraries
  - What form of classes?
The general pattern for a searching routine:

\[
\text{has } (t: \text{TABLE}; x: \text{ELEMENT}): \text{BOOLEAN is}
\]
-- Does item x appear in table t?

\[
\text{local pos: POSITION}
\]
\[
\text{do}
\]
\[
\text{from}
\]
\[
pos := \text{initial\_position } (t, x)
\]
\[
\text{until}
\]
\[
\text{exhausted } (t, pos) \text{ or else found } (t, x, pos)
\]
\[
\text{loop}
\]
\[
pos := \text{next } (t, x, pos)
\]
\[
\text{end}
\]
\[
\text{Result := found } (t, x, pos)
\]
\[
\text{end}
\]
Issues for a general searching module

- **Type variation:**
  - What are the table elements?

- **Routine grouping:**
  - A searching routine is not enough: it should be coupled with routines for table creation, insertion, deletion etc.

- **Implementation variation:**
  - Many possible choices of data structures and algorithms: sequential table (sorted or unsorted), array, binary search tree, file, ...
Issues

- Representation independence:
  - Can a client request an operation such as table search *(has)* without knowing what implementation is used internally?

\[ has (t1, y) \]
Issues

- Factoring out commonality:
  - How can the author of supplier modules take advantage of commonality within a subset of the possible implementations?
  - Example: the set of sequential table implementations.
  - A common routine text for \textit{has}:

\begin{verbatim}
has (....; x: T): BOOLEAN is
  -- Does x appear in the table?
  do
    from start until after or else found (x) loop
      forth
    end
  end
Result := found (x)
end
\end{verbatim}
Factoring out commonality

before

1

start

item

back

index

forth

count

after
Factoring out commonality

TABLE has

SEQUENTIAL_TABLE

ARRAY_TABLE

LINKED_TABLE

FILE_TABLE

TREE_TABLE

HASH_TABLE

start after found forth
## Implementation variants

<table>
<thead>
<tr>
<th></th>
<th>start</th>
<th>forth</th>
<th>after</th>
<th>found ( (x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Array</strong></td>
<td>( i := 1 )</td>
<td>( i := i + 1 )</td>
<td>( i &gt; \text{count} )</td>
<td>( t[i] = x )</td>
</tr>
<tr>
<td><strong>Linked list</strong></td>
<td>( c := \text{first Cell} )</td>
<td>( c := c.right )</td>
<td>( c = \text{Void} )</td>
<td>( c.item = x )</td>
</tr>
<tr>
<td><strong>File</strong></td>
<td>( \text{rewind} )</td>
<td>( \text{read} )</td>
<td>( \text{end of file} )</td>
<td>( f = \xi )</td>
</tr>
</tbody>
</table>
Encapsuation languages ("Object-based")

- Ada, Modula-2, CLU...

- **Basic idea:** gather a group of routines serving a related purpose, such as *has*, *insert*, *remove* etc., together with the appropriate data structure descriptions.

- This addresses the Related Routines issue.

- **Advantages:**
  - For supplier author: Get everything under one roof. Simplifies configuration management, change of implementation, addition of new primitives.
  - For client author: Find everything at one place. Simplifies search for existing routines, requests for extensions.
Complementary material

- OOSC2:
  - Chapter 3: Modularity
  - Chapter 4: Approaches to reusability
End of lecture 2