Software Architecture

Bertrand Meyer
ETH Zurich, April-July 2006

Lecture 2:
Modularity; abstract data types
Reading assignment for this week

OOSC, chapters
  3: Modularity
  6: Abstract data types

In particular pp.153-159, sufficient completeness
Modularity

General goal:

Ensuring that software systems are structured into units (modules) chosen to favor

- Extendibility
- Reusability
- “Maintainability”
- Other benefits of clear, well-defined architectures
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

C

D

C1

I

L1

L2

I1

C2

Loop

Sequence

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.

(A)  
(B)  
(C)
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 $x$
  - Modifies
  - Accesses
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

\[ \text{balance} = \text{list\_of\_deposits\_total} - \text{list\_of\_withdrawals\_total} \]

(A1) \hspace{1cm} \text{list\_of\_deposits} \hspace{1cm} \text{list\_of\_withdrawals} \hspace{1cm} \text{balance}

(A2) \hspace{1cm} \text{list\_of\_deposits} \hspace{1cm} \text{list\_of\_withdrawals}

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

Simula, Eiffel: 
\text{a.balance}

balance (a) \hspace{1cm} 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.
Information Hiding Principle

- 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
An object has an interface
An object has an implementation
Information hiding

Chair of Software Engineering

Software architecture
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.
The Open-Closed principle
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: Technical issues

General pattern for a searching routine:

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

\[
\text{local}
\]

\[
\text{pos: POSITION}
\]

\[
\text{do}
\]

\[
\text{from}
\]

\[
\text{pos := initial\_position } (t, x)
\]

\[
\text{until}
\]

\[
\text{exhausted } (t, \text{pos}) \text{ or else found } (t, x, \text{pos})
\]

\[
\text{loop}
\]

\[
\text{pos := next } (t, x, \text{pos})
\]

\[
\text{end}
\]

\[
\text{Result := found } (t, x, \text{pos})
\]

\[
\text{end}
\]
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 \( (\text{has}) \) without knowing what implementation is used internally?

  \[
  \text{has} \ (t1, \ y)
  \]
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 *has*:

  ```
  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
  ```
Factoring out commonality

- TABLE
  - has
    - SEQUENTIAL_TABLE
    - TREE_TABLE
    - HASH_TABLE
      - FILE_TABLE
      - LINKED_TABLE
      - ARRAY_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>Array</td>
<td>( i := 1 )</td>
<td>( i := i + 1 )</td>
<td>( i &gt; \text{count} )</td>
<td>( t [i] = x )</td>
</tr>
<tr>
<td>Linked list</td>
<td>( c := \text{first}_\text{cell} )</td>
<td>( c := c.\text{right} )</td>
<td>( c = \text{Void} )</td>
<td>( c.\text{item} = x )</td>
</tr>
<tr>
<td>File</td>
<td>( \text{rewind} )</td>
<td>( \text{read} )</td>
<td>( \text{end}<em>\text{of}</em>\text{file} )</td>
<td>( f\uparrow = \xi )</td>
</tr>
</tbody>
</table>
Encapsulation languages ("Object-based")

Ada, Modula-2, Oberon, 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.
Abstract Data Types (ADT)

- Why use the objects?
- The need for data abstraction
- Moving away from the physical representation
- Abstract data type specifications
- Applications to software design
The first step

- A system performs certain actions on certain data.
- Basic duality:
  - Functions [or: Operations, Actions]
  - Objects [or: Data]
Finding the structure

- The structure of the system may be deduced from an analysis of the functions (1) or the objects (2).

- Resulting analysis and design method:
  - Process-based decomposition: classical (routines)
  - Object-oriented decomposition
Arguments for using objects

- **Reusability**: Need to reuse whole data structures, not just operations
- **Extendibility, Continuity**: Objects remain more stable over time.

```
Employee information

Hours worked

Produce Paychecks

Paychecks
```
Object-oriented software construction is the approach to system structuring that bases the architecture of software systems on the types of objects they manipulate — not on “the” function they achieve.
The O-O designer’s motto

- Ask NOT first WHAT the system does:

  Ask WHAT it does it TO!
Issues of object-oriented design

- How to find the object types.
- How to describe the object types.
- How to describe the relations and commonalities between object types.
- How to use object types to structure programs.
Description of objects

- Consider not a single object but a type of objects with similar properties.

- Define each type of objects not by the objects’ physical representation but by their behavior: the services (FEATURES) they offer to the rest of the world.

- External, not internal view: ABSTRACT DATA TYPES
The theoretical basis

- The main issue: How to describe program objects (data structures):
  - Completely
  - Unambiguously
  - Without overspecifying?
    (Remember information hiding)
A stack, concrete object

“Push” $x$ on stack $\text{representation}$:

$$\text{representation}[\text{count}] := x$$
$$\text{count} := \text{count} + 1$$
A stack, concrete object

“Push” $x$ on stack $\text{representation}$:

\[
\text{representation}[\text{count}] := x \\
\text{count} := \text{count} + 1
\]

“Push” $x$ on stack $\text{representation}$:

\[
\text{representation}[\text{free}] := x \\
\text{free} := \text{free} - 1
\]
A stack, concrete object

“Push” $x$ on stack representation:

```
representation[count] := x
count := count + 1
```

“Push” $x$ on stack representation:

```
representation[free] := x
free := free - 1
```

“Push” operation:

```
new (n)
n.item := x
n.previous := last
head := n
```
Stack: An Abstract Data Type (ADT)

- **Types:**
  
  $\text{STACK } [G]$
  
  -- $G$: Formal generic parameter

- **Functions (Operations):**
  
  $\text{put: STACK } [G] \times G \rightarrow \text{STACK } [G]$
  
  $\text{remove: STACK } [G] \rightarrow \text{STACK } [G]$
  
  $\text{item: STACK } [G] \rightarrow G$
  
  $\text{empty: STACK } [G] \rightarrow \text{BOOLEAN}$
  
  $\text{new: STACK } [G]$
Using functions to model operations

\[ \text{put}(s, x) = s' \]
Reminder: Partial functions

- A partial function, identified here by $\rightarrow$, is a function that may not be defined for all possible arguments.

- Example from elementary mathematics:
  - inverse: $\mathbb{R} \rightarrow \mathbb{R}$, such that

$$\text{inverse} \ (x) = \frac{1}{x}$$
The STACK ADT (continued)

- **Preconditions:**
  
  \[
  \text{remove} \ (s: \text{STACK} \ [G]) \ \text{require not empty} \ (s) \\
  \text{item} \ (s: \text{STACK} \ [G]) \ \text{require not empty} \ (s)
  \]

- **Axioms:** For all \(x: G, s: \text{STACK} \ [G]\)
  
  \[
  \text{item} \ (\text{put} \ (s, x)) = x \\
  \text{remove} \ (\text{put} \ (s, x)) = s \\
  \text{empty} \ (\text{new}) \\
  \quad \text{(or: empty} \ (\text{new}) = \text{True}) \\
  \text{not empty} \ (\text{put} \ (s, x)) \\
  \quad \text{(or: empty} \ (\text{put} \ (s, x)) = \text{False})
  \]
Adapt the preceding specification of stacks (LIFO, Last-In First-Out) to describe queues instead (FIFO).

Adapt the preceding specification of stacks to account for bounded stacks, of maximum size capacity.

- Hint: put becomes a partial function.
Formal stack expressions

\[
\text{value} = \text{item} \ (\text{remove} \ (\text{put} \ (\text{remove} \ (\text{put} \ (\text{put} \ (\text{put} \ (\text{new}, x_8), x_7), x_6), \text{item} \\
\text{remove} \ (\text{put} \ (\text{put} \ (\text{new}, x_5), x_4)))))), x_2)), x_1))
\]
Expressed differently

\[
\text{value} = \text{item} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{put} (\text{put} (\text{put} (\text{put} (\text{put} (\text{new}, x_8), x_7), x_6)))))
\]

\[
\text{item} (\text{remove} (\text{put} (\text{put} (\text{put} (\text{new}, x_5), x_4))))), x_2)), x_1))))
\]

- \( s_1 = \text{new} \)
- \( s_2 = \text{put} (\text{put} (\text{put} (s_1, x_8), x_7), x_6) \)
- \( s_3 = \text{remove} (s_2) \)
- \( s_4 = \text{new} \)
- \( s_5 = \text{put} (\text{put} (s_4, x_5), x_4) \)
- \( s_6 = \text{remove} (s_5) \)
- \( y_1 = \text{item} (s_6) \)
- \( s_7 = \text{put} (s_3, y_1) \)
- \( s_8 = \text{put} (s_7, x_2) \)
- \( s_9 = \text{remove} (s_8) \)
- \( s_{10} = \text{put} (s_9, x_1) \)
- \( s_{11} = \text{remove} (s_{10}) \)

- \( \text{value} = \text{item} (s_{11}) \)
value = item (  
  remove (  
    put (  
      remove (  
        put (  
          remove (  
            put (  
              remove (  
                put (  
                  new, x8), x7), x6)  
            , item (  
              remove (  
                put (  
                  new, x5), x4)  
            )  
          , x2)  
        )  
      )  
    )  
  )  
)
value = item (  
    remove (  
        put (  
            remove (  
                put (  
                    remove (  
                        put (put (put (new, x8), x7), x6)  
                    , item (  
                        remove (  
                            put (put (new, x5), x4)  
                        , x2)  
                    , x1)  
                )  
            )  
        )  
    )  
)
Expression reduction

```
value = item (remove (put (remove (put (remove (put (put (new, x8), x7), x6)
                                 , item (remove (put (put (new, x5), x4)
                                               )
                                         , x2)
                                   , x1))
                 )
         )
```
Expression reduction

\[
\text{value} = \text{item} \left( \begin{array}{c}
\text{remove} \\
\text{put} \\
\text{put} \\
\text{put} \\
\text{remove} \\
\text{put} \\
\text{put} \\
\text{put} \\
\text{new}, x_8 \\
\text{new}, x_5 \\
\text{new}, x_2 \\
\text{new}, x_1
\end{array} \right)
\]

Stack 1

\[
x_6 \\
x_7 \\
x_8
\]
Expression reduction

value = item (remove (put (remove (put (put (new, x8), x7), x6)
  , item (remove (put (put (new, x5), x4)
          , x2)
        , x1)
    )
  , x8)
)
value = item (remove (put (remove (put (put (new, x8), x7), x6) remove (put (put (new, x5), x4) ), x2) , x1) , item (remove (put (new, x8), x7), x6) )
Expression reduction

\[
\text{value} = \text{item (}
    \text{remove (}
        \text{put (}
            \text{remove (}
                \text{put (}
                    \text{put (}
                        \text{remove (}
                            \text{put (}
                                \text{put (}
                                    \text{new, x8), x7), x6)
                            )}
                        )}
                    )}
                )}
            )}
        )}
    )}
\]
value = item ( remove ( put ( remove ( put ( put ( remove ( put ( put ( new, x8), x7), x6) , item ( remove ( put ( new, x5), x4) , x2) , x1) , x2) , x1) , x2) , x1) )
value = item (remove (put (remove (put (put (new, x8), x7), x6) remove (put (put (new, x5), x4) put (x2), item), x1) x7)) Stack 1

Stack 2
Expression reduction

\[
value = item \left(\begin{array}{c}
remove \\
put \\
remove \\
put \\
put
\end{array}\right)
\]

\[
remove \left(\begin{array}{c}
put \\
put \\
put
\end{array}\right)
\]

\[
remove \left(\begin{array}{c}
put (put (new, x8), x7), x6
\end{array}\right)
\]

\[
, item \left(\begin{array}{c}
remove \\
put (put (new, x5), x4)
\end{array}\right)
\]

\[
, item \left(\begin{array}{c}
remove \\
put (put (new, x5), x4)
\end{array}\right)
\]

\[
, x2)
\]

\[
, x1)
\]

\[
, x1)
\]

Stack 1

Stack 2
Expression reduction

\[
\text{value} = \text{item (}
    \text{remove (}
        \text{put (}
            \text{remove (}
                \text{put (}
                    \text{put (}
                        \text{put (}
                            \text{put (}
                                \text{new, } x_8
                            , x_7
                        , x_6
                    , x_5
                , x_4
            , x_2
        , x_1
    , x_0
)}\]
Expression reduction

\[
\text{value} = \text{item} \left( \text{remove} \left( \text{put} \left( \text{remove} \left( \text{put} \left( \text{remove} \left( \text{put} \left( \text{put} \left( \text{new}, x_8 \right), x_7 \right), x_6 \right) \right), \text{item} \left( \text{remove} \left( \text{put} \left( \text{put} \left( \text{new}, x_5 \right), x_4 \right) \right), x_2 \right), x_1 \right) \right) \right) \right)
\]
Expression reduction

value = item (remove (put (remove (put (remove (put (put (put (new, x8), x7), x6), item (remove (put (put (new, x5), x4), x2) , x2), x1)), x2))) , x2), x5, x7, x8) Stack 1 Stack 2

Chair of Software Engineering

Software Architecture
Expression reduction

\[ \text{value} = \text{item} ( \text{remove} ( \text{put} ( \text{remove} ( \text{put} \text{item} ) ) ) ) \]

Stack 1: Stack 2

\[ x1 \]
\[ x5 \]
\[ x7 \]
\[ x8 \]
value = item (remove (put (remove (put (put (put (new, x8), x7), x6)
, item (remove (put (put (new, x5), x4)
, x2)
, x1))
, x2)
, x1)
Expression reduction

value = item (remove (put (remove (put (remove (put (put (put (new, x8), x7), x6)
          , item (remove (put (put (new, x5), x4)
                        , x2)
                    , x1)
                )
            )
        )
    )
  )
)
Expressed differently

\[ value = \text{item} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{put} (\text{put} (\text{put} (\text{put} (\text{new}, x_8), x_7), x_6)), \text{item} (\text{remove} (\text{put} (\text{put} (\text{new}, x_5), x_4))), x_2), x_1))) \]

- \( s_1 = \text{new} \)
- \( s_2 = \text{put} (\text{put} (s_1, x_8), x_7), x_6) \)
- \( s_3 = \text{remove} (s_2) \)
- \( s_4 = \text{new} \)
- \( s_5 = \text{put} (\text{put} (s_4, x_5), x_4) \)
- \( s_6 = \text{remove} (s_5) \)
- \( y_1 = \text{item} (s_6) \)
- \( s_7 = \text{put} (s_3, y_1) \)
- \( s_8 = \text{put} (s_7, x_2) \)
- \( s_9 = \text{remove} (s_8) \)
- \( s_{10} = \text{put} (s_9, x_1) \)
- \( s_{11} = \text{remove} (s_{10}) \)
- \( value = \text{item} (s_{11}) \)
An operational view of the expression

\[
value = item(\text{remove}(\text{put}(\text{remove}(\text{put}(\text{put}(\text{put}(\text{put}(\text{new}, x8), x7), x6)), \text{item}(\text{remove}(\text{put}(\text{put}(\text{new}, x5), x4)))), x2)), x1))
\]
Is my specification complete?

- Intuitively clear: captures all that’s needed
- In practice: complete with respect to what?
Properties of an ADT specification

Specification for an ADT $T$ is:

- **Consistent** if the axioms do not lead to a contradiction

- **Sufficiently complete** if any query expression that is correct may be reduced through application of the axioms to a form not involving $T$
Correct expression

An expression built from an ADT specification is correct if arguments to all functions satisfy their preconditions.
Sufficient completeness

- Three forms of functions in specification of $T$:
  - Creators:
    \[ \text{OTHER} \rightarrow T \]  
    e.g. \textit{new}
  - Queries:
    \[ T \times \ldots \rightarrow \text{OTHER} \]  
    e.g. \textit{item, empty}
  - Commands:
    \[ T \times \ldots \rightarrow T \]  
    e.g. \textit{put, remove}

- \textbf{Query expression}: outermost function is a query
- \textbf{Sufficiently complete} specification: any such expression can be expressed without reference to $T$. 
Stack: An Abstract Data Type

- **Types:**
  
  \[ \text{STACK} \ [G] \]
  
  \[ \text{-- } G: \text{ Formal generic parameter} \]

- **Functions (Operations):**
  
  \[ \text{put: } \text{STACK} \ [G] \times G \rightarrow \text{STACK} \ [G] \]
  
  \[ \text{remove: } \text{STACK} \ [G] \rightarrow \text{STACK} \ [G] \]
  
  \[ \text{item: } \text{STACK} \ [G] \rightarrow G \]
  
  \[ \text{empty: } \text{STACK} \ [G] \rightarrow \text{BOOLEAN} \]
  
  \[ \text{new: } \text{STACK} \ [G] \]
Abstract data types provide an ideal basis for modularizing software.

- Build each module as an *implementation* of an ADT:
  - Implements a set of *objects* with same *interface*
  - Interface is defined by a set of operations (the ADT’s functions) constrained by abstract properties (its axioms and preconditions).

- The module consists of:
  - A *representation* for the ADT
  - An *implementation* for each of its operations
  - Possibly, auxiliary operations
Implementing an ADT

- Three components:
  (E1) The ADT’s specification: functions, axioms, preconditions.
    (Example: stacks.)
  (E2) Some representation choice.
    (Example: \(<representation, count>\).)
  (E3) A set of subprograms (routines) and attributes, each implementing one of the functions of the ADT specification (E1) in terms of the chosen representation (E2).
    (Example: routines \(put, remove, item, empty, new\).)
A choice of stack representation

```
"Push" operation:

\[
\text{count} := \text{count} + 1
\]

\[
\text{representation}[\text{count}] := x
\]
```
Application to information hiding

Public part:
ADT specification (*E1*)

Secret part:

- Choice of representation (*E2*)
- Implementation of functions by features (*E3*)
Object technology: A first definition

- Object-oriented software construction is the approach to system structuring that bases the architecture of software systems on the types of objects they manipulate — not on “the” function they achieve.
Object-oriented software construction is the construction of software systems as structured collections of (possibly partial) abstract data type implementations.
Classes: The fundamental structure

- Merging of the notions of **module** and **type**:
  - Module = Unit of decomposition: set of services
  - Type = Description of a set of run-time objects ("instances" of the type)

- The connection:
  - The services offered by the class, viewed as a module, are the operations available on the instances of the class, viewed as a type.
Class relations

- Two relations:
  - Client
  - Heir
deferred class

<stdlib>

COUNTER

feature

item: INTEGER is
  -- Counter value
  deferred
end

up is
  -- Increase item by 1.
  deferred
  ensure
  item = old item + 1
end

down is
  -- Decrease item by 1.
  deferred
  ensure
  item = old item - 1
end

invariant
  item >= 0
end
End of lecture 2