Software Architecture

Bertrand Meyer

ETH Zurich, March-July 2007

Lecture 2: Modularity and 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:**

Ensure 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

The 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

C1

I

D

I1

C2

I2

C

Conditional

Sequence

Loop
Top-down design


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

The method favors the production of software elements that may be freely combined with each other to produce new software.

Example: Unix shell conventions
Program1 | Program2 | Program3
The method yields software systems whose modular structure remains compatible with any modular structure devised in the process of modeling the problem domain.
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
Whenever two modules communicate, this is clear from the text of one or both of them.

A

\textit{modifies}

Data item $x$

\textit{accesses}

B
Continuity

The 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

It doesn't matter to the client whether you look up or compute

A call such as

`your_account.balance`

could use an attribute or a function
Uniform Access

balance = list_of_deposits.total - list_of_withdrawals.total

\[ \text{(A1)} \]

\[ \text{list_of_deposits} \]
\[ \text{list_of_withdrawals} \]
\[ \text{balance} \]

\[ \text{(A2)} \]

\[ \text{list_of_deposits} \]
\[ \text{list_of_withdrawals} \]

Ada, Pascal, C/C++, Java, C#: a.balance
balance(a)

Simula, Eiffel: a.balance
a.balance()
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.
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

Justifications:

- Continuity
- Decomposability
An object has an interface
An object has an implementation
Information hiding
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.)
General pattern for a searching routine:

\[ \text{has}(t, \text{ TABLE }, x, \text{ ELEMENT }): \text{ BOOLEAN} \text{ is} \]

-- Does item \( x \) appear in table \( t \)?

\[
\text{local} \quad pos: \text{ POSITION} \\
\text{do} \\
\text{from} \\
\quad pos := \text{ initial\_position}(t, x) \\
\text{until} \\
\quad \text{exhausted}(t, pos) \text{ or else found}(t, x, pos) \\
\text{loop} \\
\quad pos := \text{ next}(t, x, pos) \\
\text{end} \\
\text{Result} := \text{ found}(t, x, pos) \\
\text{end} \]
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 \( (\text{has}) \) without knowing what implementation is used internally?

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

ARRAY_TABLE

start after found forth

LINKED_TABLE

TREE_TABLE

FILE_TABLE

HASH_TABLE
## 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.\text{right})</td>
<td>(c = \text{Void})</td>
<td>(c.\text{item} = x)</td>
</tr>
<tr>
<td><strong>File</strong></td>
<td>\textit{rewind}</td>
<td>\textit{read}</td>
<td>\textit{end_of_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.
The concept of Abstract Data Type (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 architectural style and analysis/design method:

- (1) Top-down, functional decomposition
- (2) Object-oriented
Arguments for using objects

**Reusability:** Need to reuse whole data structures, not just operations

**Extendibility, Continuity:** Object categories remain more stable over time.

```
Employee information

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Produce</td>
</tr>
<tr>
<td>Paychecks</td>
</tr>
</tbody>
</table>

Paychecks

Hours worked
```
Object-oriented software construction is the software architecture method that bases the structure of systems on the types of objects they handle — not on “the” function they achieve.
Ask not first WHAT the system does:

Ask WHAT it does it to!
Issues of object-oriented architecture

- 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)
Abstract Data Types

A formal way of describing data structures

Benefits:

- **Modular, precise description of a wide range of problems**
- Enables proofs
- Basis for object technology
- Basis for object-oriented requirements
A stack, concrete object

Representing "Array Up":

Implementing a "PUSH" operation:

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

Representation 1: “Array Up”

Representation 2: “Array Down”

Implementing a “PUSH” operation:

\[
\begin{align*}
\text{count} & := \text{count} + 1 \\
\text{rep}[\text{count}] & := x \\
\text{rep}[\text{free}] & := x \\
\text{free} & := \text{free} - 1
\end{align*}
\]
A stack, concrete object

Implementing a “PUSH” operation:

Representation 1: “Array Up”
rep[capacity] := x
count := count + 1
rep[free] := x
free := free - 1
create cell
head := cell

Representation 2: “Array Down”

Representation 3: “Linked List”

Stack: An Abstract Data Type (ADT)

Types:

\[ \text{STACK}[G] \]

--- \( G \): Formal generic parameter

Functions (Operations):

- \( \text{put} : \text{STACK}[G] \times G \rightarrow \text{STACK}[G] \)
- \( \text{remove} : \text{STACK}[G] \rightarrow \text{STACK}[G] \)
- \( \text{item} : \text{STACK}[G] \rightarrow G \)
- \( \text{empty} : \text{STACK}[G] \rightarrow \text{BOOLEAN} \)
- \( \text{new} : \text{STACK}[G] \)
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:

- \textit{inverse}: $\mathbb{R} \rightarrow \mathbb{R}$, such that

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

Preconditions:

\[
\begin{align*}
\text{remove} (s : \text{STACK} [G]) & \text{ require not empty} (s) \\
\text{item} (s : \text{STACK} [G]) & \text{ require not empty} (s)
\end{align*}
\]

Axioms: For all \( x : G, s : \text{STACK} [G]\)

\[
\begin{align*}
\text{item} (\text{put} (s, x)) & = x \\
\text{remove} (\text{put} (s, x)) & = s \\
\text{empty} (\text{new}) & \\
& \text{(can also be written: empty} (\text{new}) = \text{True)}
\end{align*}
\]

\[
\begin{align*}
\text{not empty} (\text{put} (s, x)) & \\
& \text{(can also be written: empty} (\text{put} (s, x)) = \text{False)}
\end{align*}
\]
Exercises

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

\[
\text{value} =
\text{item} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{put} (\text{put} (\text{put} \\
\text{remove} (\text{put} (\text{put} (\text{new}, x^8), x^7), x^6)), \\
\text{item} (\text{remove} (\text{put} (\text{put} (\text{new}, x^5), x^4)))))}, \\
x^2)), x^1)))
\]
Expressed differently

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

\[
s1 = \text{new}
\]
\[
s2 = \text{put} \ (\text{put} \ (\text{put} \ (s1, x8), x7), x6)
\]
\[
s3 = \text{remove} \ (s2)
\]
\[
s4 = \text{new}
\]
\[
s5 = \text{put} \ (\text{put} \ (s4, x5), x4)
\]
\[
s6 = \text{remove} \ (s5)
\]
\[
y1 = \text{item} \ (s6)
\]
\[
s7 = \text{put} \ (s3, y1)
\]
\[
s8 = \text{put} \ (s7, x2)
\]
\[
s9 = \text{remove} \ (s8)
\]
\[
s10 = \text{put} \ (s9, x1)
\]
\[
s11 = \text{remove} \ (s10)
\]
\[
\text{value} = \text{item} \ (s11)
\]
Expression reduction

value = item

Stack 1

new, x8, x7, x6

put (put (put (new, x5), x4)

put (put (put new, x8), x7, x6)

remove

put

remove

put

remove

put

remove

put
Expression reduction

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

Stack 1

x8
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(
          remove(
            put(item(
              remove(
                put(
                  new, x5)
                x4
              )
            x6)
          )
        x7)
      x8
    )
  )
)
Expression reduction

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

Stack 1

x6
x7
x8
Expression reduction

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

Stack 1
Stack 2

x7
x8
Expression reduction

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

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

Stack 1

Stack 2
Expression reduction

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

Stack 1

Stack 2

- x7
- x8
- x4
- x5
Expression reduction

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

Stack 1
Stack 2
Expression reduction

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

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

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

Stack 1

\[
\text{Stack 2}
\]

x2
x5
x7
x8
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)
\text{, item (}
\text{remove (}
\text{put (put (new, x5), x4)
\text{, x2)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
\text{, x1)
Expression reduction

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

Stack 2
Expression reduction

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

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

\[
\begin{align*}
s_1 &= \text{new} \\
s_2 &= \text{put} (\text{put} (\text{put} (s_1, x_8), x_7), x_6) \\
s_3 &= \text{remove} (s_2) \\
s_4 &= \text{new} \\
s_5 &= \text{put} (\text{put} (s_4, x_5), x_4) \\
s_6 &= \text{remove} (s_5)
\end{align*}
\]

\[
\begin{align*}
y_1 &= \text{item} (s_6) \\
s_7 &= \text{put} (s_3, y_1) \\
s_8 &= \text{put} (s_7, x_2) \\
s_9 &= \text{remove} (s_8) \\
s_{10} &= \text{put} (s_9, x_1) \\
s_{11} &= \text{remove} (s_{10}) \\
\text{value} &= \text{item} (s_{11})
\end{align*}
\]
An operational view of the expression

\[
\text{value} = \text{item (remove (put (remove (put (put (put (put (put (put (put (new, x8), x7), x6)))) (s9, s11)), x8))))}
\]
Sufficient completeness

Three forms of functions in the specification of an ADT $T$:

- **Creators:**
  
  $\text{OTHER} \rightarrow T$
  
  e.g. `new`

- **Queries:**
  
  $T \times \ldots \rightarrow \text{OTHER}$
  
  e.g. `item`, `empty`

- **Commands:**
  
  $T \times \ldots \rightarrow T$
  
  e.g. `put`, `remove`

**Sufficiently Complete specification**

An ADT specification with axioms that make it possible to reduce any “Query Expression” of the form $f(\ldots)$

where $f$ is a query, to a form not involving $T$
The stack example

Types

\[ \text{STACK}[G] \]

Functions

\[
\begin{align*}
\text{put} : \text{STACK}[G] \times G & \rightarrow \text{STACK}[G] \\
\text{remove} : \text{STACK}[G] & \rightarrow \text{STACK}[G] \\
\text{item} : \text{STACK}[G] & \rightarrow G \\
\text{empty} : \text{STACK}[G] & \rightarrow \text{BOOLEAN} \\
\text{new} : \text{STACK}[G] &
\end{align*}
\]
ADTs and software architecture

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: \(<rep, count>\))

(E3) A set of subprograms (routines) and attributes, each implementing one of the functions of the ADT specification (E1) in terms of chosen representation (E2)
(Example: routines put, remove, item, empty, new)
A choice of stack representation

“Push” operation:

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

\[ \text{rep}[\text{count}] := x \]
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.
Applying ADTs to information hiding

Public part:
- ADT specification \((E1)\)

Secret part:
- Choice of representation \((E2)\)
- Implementation of functions by features \((E3)\)
Object-oriented software construction is the software architecture method that bases the structure of systems on the types of objects they handle — not on “the” function they achieve.
A more precise definition

Object-oriented software construction is the construction of software systems as structured collections of (possibly partial) abstract data type implementations.
The fundamental structure: the class

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*
Overall system structure

CHUNK
- space_before
- space_after
- add_space_before
- add_space_after
- word_count
- justified

PARAGRAPH
- add_word
- remove_word
- justify
- unjustify

WORD
- length
- font
- set_font
- hyphenate_on
- hyphenate_off

FEATURES

QUERIES

COMMANDS

Inheritance

Client
End of lecture 2
Bounded stacks

Types:

\[ \text{BSTACK}[G] \]

Functions (Operations):

\[ \text{put}: \text{BSTACK}[G] \times G \rightarrow \text{BSTACK}[G] \]
\[ \text{remove}: \text{BSTACK}[G] \rightarrow \text{BSTACK}[G] \]
\[ \text{item}: \text{BSTACK}[G] \rightarrow G \]
\[ \text{empty}: \text{BSTACK}[G] \rightarrow \text{BOOLEAN} \]
\[ \text{new}: \text{BSTACK}[G] \]
\[ \text{capacity}: \text{BSTACK}[G] \rightarrow \text{INTEGER} \]
\[ \text{count}: \text{BSTACK}[G] \rightarrow \text{INTEGER} \]
\[ \text{full}: \text{BSTACK}[G] \rightarrow \text{BOOLEAN} \]
Bounded stacks (continued)

Preconditions:

\[ \text{remove} (s : BSTACK[G]) \text{ require not empty} (s) \]
\[ \text{item} (s : BSTACK[G]) \text{ require not empty} (s) \]
\[ \text{put} (s : BSTACK[G]) \text{ require not full} (s) \]

Axioms: For all \( x : G, s : BSTACK[G] \)

\[ \text{item} (\text{put} (s, x)) = x \]
\[ \text{remove} (\text{put} (s, x)) = s \]
\[ \text{empty} (\text{new}) \]
\[ \text{not empty} (\text{put} (s, x)) \]
\[ \text{full} = (\text{count} = \text{capacity}) \]
\[ \text{count} (\text{new}) = 0 \]
\[ \text{count} (\text{put} (s, x)) = \text{count} (s) + 1 \]
\[ \text{count} (\text{remove} (s)) = \text{count} (s) - 1 \]