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
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.

![Graphs showing (A), (B), and (C)]

### Small interfaces principle

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

![Diagram showing modules A and B exchanging x, y, and z]

### Explicit interfaces principle

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

![Diagram showing Module A modifying Data item x and Module B accessing x]
Continuity

- 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

balance = list_of_deposits.total - list_of_withdrawals.total

(A1) list_of_deposits
   list_of_withdrawals
   balance

(A2) list_of_deposits
   list_of_withdrawals

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

- Justifications:
  - Continuity
  - Decomposability

An object has an interface

An object has an implementation
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 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:

```plaintext
has (t: TABLE; x: ELEMENT): BOOLEAN is
  -- Does item x appear in table t?
  local
    pos: POSITION
    do
      from pos := initial_position (t, x)
      until exhausted (t, pos) or else found (t, x, pos)
      loop
        pos := next (t, x, pos)
      end loop
    pos := next (t, x, pos)
  end do
result := found (t, x, pos)
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?
    \[
    \text{has}(t_1, y)
    \]

- 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 \( \text{has} \):
    \[
    \text{has} \quad \ldots; 
    x: T 
    \]
    \[
    \text{BOOLEAN is}
    \]
    \[
    \text{do}
    \]
    \[
    \text{from} \quad \text{start} \quad \text{until} \quad \text{after} \quad \text{or else} \quad \text{found} \quad (x) \quad \text{loop}
    \]
    \[
    \text{forkth}
    \]
    \[
    \text{end}
    \]
    \[
    \text{forth}
    \]
    \[
    \text{Result} := \text{found} \quad (x)
    \]
    \[
    \text{end}
    \]

Factoring out commonality

- TABLE
  - has
  - start
  - after
  - found
  - forth
  - SEQUENTIAL
    - TABLE
  - TREE
    - TABLE
  - HASH
    - TABLE
  - ARRAY
    - TABLE
  - LINKED
    - TABLE
  - FILE
    - 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>Array</td>
<td>c := 1</td>
<td>i := i + 1</td>
<td>i &gt; count</td>
<td>i | x</td>
</tr>
<tr>
<td>Linked list</td>
<td>c := first-cell</td>
<td>c := c.right</td>
<td>c := Void</td>
<td>c.item = x</td>
</tr>
<tr>
<td>File</td>
<td>rewind</td>
<td>read</td>
<td>end_of_file</td>
<td>i := i</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]

```
Actions ------ Objects

Processor
```

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 ------ Produce Paychecks ------ Paychecks

Hours worked
```
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.

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

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

```
<table>
<thead>
<tr>
<th>capacity</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_up</td>
<td></td>
</tr>
</tbody>
</table>
```
A stack, concrete object

- capacity
- count
- representation
- "Push" x on stack representation:
  - representation [count] := x
  - count := count + 1

- free
- "Push" x on stack representation:
  - representation [free] := x
  - free := free - 1

Stack: An Abstract Data Type (ADT)

- Types:
  - \( \text{STACK} \ [G] \rightarrow G \): Formal generic parameter

- Functions (Operations):
  - put: \( \text{STACK} \ [G] \times G \rightarrow \text{STACK} \ [G] \)
  - remove: \( \text{STACK} \ [G] \rightarrow \text{STACK} \ [G] \)
  - item: \( \text{STACK} \ [G] \rightarrow G \)
  - empty: \( \text{STACK} \ [G] \rightarrow \text{BOOLEAN} \)
  - 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:
  - inverse: \( \mathbb{R} \rightarrow \mathbb{R} \), such that
  \[
  \text{inverse}(x) = 1 / x
  \]

The STACK ADT (continued)

- Preconditions:
  \[
  \text{remove}\ (s: \text{STACK}\ [G]) \require\ \text{not empty}\ (s)
  \]
  \[
  \text{item}\ (s: \text{STACK}\ [G]) \require\ \text{not empty}\ (s)
  \]
- Axioms: For all \( x: G, s: \text{STACK}\ [G] \)
  \[
  \text{item}\ (\text{put}\ (s, x)) = x
  \]
  \[
  \text{remove}\ (\text{put}\ (s, x)) = s
  \]
  \[
  \text{empty}\ (\text{new})
  \]
  \[
  \text{not empty}\ (\text{put}\ (s, x))
  \]
  \[
  \text{empty}\ (\text{put}\ (s, x)) = \text{False}
  \]
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

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

Expressed differently

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

value = item {
  remove {
    put (
      remove {
        put (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 (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 (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 { put { put { put { put { put { new, x8 }, x7 }, x6 } , item { remove { put { put { new, x5 }, x4 } } } } } } } } } } , x2 }

value = item {
    remove { put {
        remove { put { put { put { put { put { put { new, x8 }, x7 }, x6 } , item { remove { put { put { new, x5 }, x4 } } } } } } } } } } , x2 }

value = item {
    remove { put {
        remove { put { put { put { put { put { put { new, x8 }, x7 }, x6 } , item { remove { put { put { new, x5 }, x4 } } } } } } } } } } , x2 }

Stack 1

Stack 2
Expression reduction

\[ \text{value} = \text{item} \{
    \text{remove} \{
        \text{put} \{
            \ldots
        \}
    \}
\}\]
Expression reduction

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

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

Expressed differently

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

An operational view of the expression

value = item (remove (put (remove (put (put (put (put (new, x8), x7), x6), item (remove (put (put (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. new
  - Queries: $T \times \ldots \rightarrow \text{OTHER}$ e.g. item, empty
  - Commands: $T \times \ldots \rightarrow T$ e.g. put, remove
- Query expression: outermost function is a query
- Sufficiently complete specification: any such expression can be expressed without reference to $T$.

Stack: An Abstract Data Type

- Types:
  $\text{STACK} \ [G]$
  $\Leftrightarrow G$: Formal generic parameter
- Functions (Operations):
  - put: $\text{STACK} \ [G] \times G \rightarrow \text{STACK} \ [G]$
  - remove: $\text{STACK} \ [G] \rightarrow \text{STACK} \ [G]$
  - item: $\text{STACK} \ [G] \rightarrow G$
  - empty: $\text{STACK} \ [G] \rightarrow \text{BOOLEAN}$
  - new: $\text{STACK} \ [G]$

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: <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

```
capacity
(array_up)
count
representation
```

"Push" operation:
- `count := count + 1`
- `representation[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 technology: More precise definition

- 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

# Overall system structure

**Class diagram**

- **Inheritance**
  - Client

**Features**

- **add_word**
- **remove_word**
- **justify**
- **unjustify**
- **length**
- **set_font**
- **hyphenate_on**
- **hyphenate_off**

**Queries**

- **space_before**
- **space_after**

**Commands**

- **add_space_before**
- **add_space_after**

---

# A very deferred class

**Deferred class**

**counter**

```plaintext
deferred class COUNTER
  feature
    item: INTEGER is -- Counter value
defered
  up is -- Increase item by 1.
  deferred
  down is -- Decrease item by 1.
  deferred
  invariant
    item >= 0
end
```

---
End of lecture 2