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

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

![Top-down functional design diagram]
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

Direct Mapping

The 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 communicate, this is clear from the text of one or both of them.

(A) (B) (C)
Continuity

The method ensures that small changes in specifications yield small changes in architecture.

Design method: Specification → 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

\[ \text{your\_account\_balance} \]

could use an attribute or a function

Uniform Access

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

(A1)

\[
\begin{array}{c}
\text{list\_of\_deposits} \\
\text{list\_of\_withdrawals} \\
\text{balance}
\end{array}
\]

(A2)

\[
\begin{array}{c}
\text{list\_of\_deposits} \\
\text{list\_of\_withdrawals}
\end{array}
\]

Ada, Pascal, C/C++, Java, C#: Simula, Eiffel:

\[ a\text{-balance} \]

\[ a\text{-balance}() \]

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.

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.

Public

Private
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 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:

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

\[
\text{-- Does item x appear in table t?}
\]

\[
\text{local pos: POSITION}
\]

\[
\text{do}
\]

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

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

\[
\text{loop}
\]

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

\[
\text{end}
\]

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

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?

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

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:

\[
\text{has (...) x: T): BOOLEAN is}
\]

\[
\text{-- Does x appear in the table?}
\]

\[
\text{do}
\]

\[
\text{from start until after or else found (x) loop}
\]

\[
\text{forth}
\]

\[
\text{end}
\]

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

end

Factoring out commonality

TABLE

has

SEQUENTIAL_TABLE

TREE_TABLE

HASH_TABLE

ARRAY_TABLE

LINKED_TABLE

FILE_TABLE

start

after

found

forth
Implementation variants

<table>
<thead>
<tr>
<th>Implementation</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; count</td>
<td>i/0 = x</td>
</tr>
<tr>
<td>Linked list</td>
<td>c := first_call</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.

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.
Object technology: A first definition

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.

The O-O designer’s motto

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

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

Implementing a “PUSH” operation:

Representation 1: "Array Up"

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

Representation 2: "Array Down"

\[ \text{free} := \text{free} - 1 \]
\[ \text{rep}[\text{free}] := x \]

Implementing a “PUSH” operation:

Representation 3: "Linked List"

\[ \text{create cell} \]
\[ \text{cell}.\text{content} := x \]
\[ \text{cell}.\text{next} := \text{last} \]
\[ \text{head} := \text{cell} \]

Stack: An Abstract Data Type (ADT)

Types:

\[ \text{STACK}\{G\} \rightarrow G : \text{Formal generic parameter} \]

Functions (Operations):

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

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:

\[ \text{inverse : } \mathbb{R} \rightarrow \mathbb{R}, \text{ such that} \]
\[ \text{inverse}(x) = 1 / x \]
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 \( x : G, s : \text{STACK}[G] \)
- \( \text{item}(\text{put}(s, x)) = x \)
- \( \text{remove}(\text{put}(s, x)) = s \)
- \( \text{empty}(\text{new}) \) (can also be written: \( \text{empty}(\text{new}) = \text{True} \))
- \( \text{not empty}(\text{put}(s, x)) \) (can also be written: \( \text{empty}(\text{put}(s, x)) = \text{False} \))

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 \( \text{capacity} \).

- Hint: \( \text{put} \) becomes a partial function.

Formal stack expressions

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

Expressed differently

\[
\begin{align*}
\text{s1} &= \text{new} \\
\text{s2} &= \text{put} (\text{put} (\text{put} (\text{new}, x_8), x_7), x_6) \\
\text{s3} &= \text{remove} (\text{s2}) \\
\text{s4} &= \text{new} \\
\text{s5} &= \text{put} (\text{put} (\text{new}, x_5), x_4) \\
\text{s6} &= \text{remove} (\text{s5}) \\
\text{s7} &= \text{put} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{put} (\text{new}, x_8), x_7), x_6), x_2), x_1)) \\
\text{s8} &= \text{put} (\text{put} (\text{new}, x_5), x_4) \\
\text{s9} &= \text{remove} (\text{s8}) \\
\text{s10} &= \text{put} (\text{s9}, x_1) \\
\text{s11} &= \text{remove} (\text{s10}) \\
\text{y1} &= \text{item} (\text{s6}) \\
\text{y2} &= \text{item} (\text{s11})
\end{align*}
\]

Expression reduction

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

Stack 1

\[
\begin{align*}
\text{s1} &= \text{new} \\
\text{s2} &= \text{put} (\text{put} (\text{put} (\text{new}, x_8), x_7), x_6) \\
\text{s3} &= \text{remove} (\text{s2}) \\
\text{s4} &= \text{new} \\
\text{s5} &= \text{put} (\text{put} (\text{new}, x_5), x_4) \\
\text{s6} &= \text{remove} (\text{s5}) \\
\text{s7} &= \text{put} (\text{remove} (\text{put} (\text{remove} (\text{put} (\text{put} (\text{new}, x_8), x_7), x_6), x_2), x_1)) \\
\text{s8} &= \text{put} (\text{put} (\text{new}, x_5), x_4) \\
\text{s9} &= \text{remove} (\text{s8}) \\
\text{s10} &= \text{put} (\text{s9}, x_1) \\
\text{s11} &= \text{remove} (\text{s10}) \\
\text{y1} &= \text{item} (\text{s6}) \\
\text{y2} &= \text{item} (\text{s11})
\end{align*}
\]

Stack 1

\[
\begin{align*}
\text{x8} \\
\text{x7} \\
\text{x6} \\
\text{x5} \\
\text{x4} \\
\text{x2} \\
\text{x1} \\
\text{x}
\end{align*}
\]
Expression reduction

\[
\text{value} = \text{item (}
\begin{aligned}
&\text{remove (}
\begin{aligned}
&\text{put (put (new, x8), x7), x6)}
\end{aligned}
\end{aligned}
\begin{aligned}
&\text{item (}
\begin{aligned}
&\text{remove (}
\begin{aligned}
&\text{put (put (new, x5), x4)}
\end{aligned}
\end{aligned}
\begin{aligned}
&\text{Stack 1 Stack 2}
\end{aligned}
\end{aligned}
\begin{aligned}
&\text{Stack 1 Stack 2}
\end{aligned}
\end{aligned}
\right)
\]
Expression reduction

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

Stack 1 Stack 2

s1 = new
s2 = put (put (s1, x8), x7), x6)
s3 = remove (s2)
s4 = new
s5 = put (put (s4, x5), x4)
s6 = remove (x5)

y1 = item (s6)
s7 = put (s3, y1)
s8 = put (s7, x2)
s9 = remove (s8)
s10 = put (s9, x1)
s11 = remove (s10)

value = item (s11)

Expressed differently

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

Stack 1 Stack 2

s2 s3 (empty)
s1

Sufficient completeness

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

- **Creators**: OTHER → T  e.g. new
- **Queries**: T × ... → OTHER  e.g. item, empty
- **Commands**: T × ... → 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(...) where f is a query, to a form not involving T

The stack example

Types

STACK[G]

Functions

remove: STACK[G] → STACK[G]
item: STACK[G] → G
empty: STACK[G] → BOOLEAN
new: 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 the 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:**
1. The ADT's specification: functions, axioms, preconditions
   - Example: stacks
2. Some representation choice
   - Example: \(<rep, count>\)
3. 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**

```
capacity
(array_up)
count
rep
```

"Push" operation:
```
count := count + 1
rep[count] := x
```

**Information hiding**

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 technology: A first definition**

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

Bounded stacks


Functions (Operations):

- **put**: $BSTACK[G] \times G \rightarrow BSTACK[G]$
- **remove**: $BSTACK[G] \rightarrow BSTACK[G]$
- **item**: $BSTACK[G] \rightarrow G$
- **empty**: $BSTACK[G] \rightarrow BOOLEAN$
- **new**: $BSTACK[G]$
- **capacity**: $BSTACK[G] \rightarrow INTEGER$
- **count**: $BSTACK[G] \rightarrow INTEGER$

- **full**: $BSTACK[G] \rightarrow BOOLEAN$

End of lecture 2
Preconditions:
- \( \text{remove} (s : BSTACK[G]) \) require not empty \( s \)
- \( \text{item} (s : BSTACK[G]) \) require not empty \( s \)
- \( \text{put} (s : BSTACK[G]) \) require not full \( s \)

Axioms: For all \( x : G, s : BSTACK[G] \)
- \( \text{item} (\text{put} (s, x)) = x \)
- \( \text{remove} (\text{put} (s, x)) = s \)
- \( \text{empty} (\text{new}) \)
- \( \neg \text{empty} (\text{put} (s, x)) \)
- \( \text{full} = (\text{count} = \text{capacity}) \)
- \( \text{count} (\text{new}) = 0 \)
- \( \text{count} (\text{put} (s, x)) = \text{count} (s) + 1 \)
- \( \text{count} (\text{remove} (s)) = \text{count} (s) - 1 \)