**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

- 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](image)
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.
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

$$\text{balance} = \text{list_of_deposits.total} - \text{list_of_withdrawals.total}$$

(A1)

(A2)

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

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 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:

```pseudocode
has (t: TABLE; x: ELEMENT): BOOLEAN is
  -- Does item x appear in table t?
  local
  do
    pos: POSITION
    from
    until
    loop
      pos := initial_position (t, x)
      exhausted (t, pos) or else found (t, x, pos)
      pos := next (t, x, pos)
    end
    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 (has) without knowing what implementation is used internally?
    ```pseudocode
    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:
    ```pseudocode
    has (....; x; T): BOOLEAN is
      -- Does x appear in the table?
      do
        from start until after or else found (x) loop
        forth
      end
      Result := found (x)
    end
    ```

Factoring out commonality
### 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>j := i + 1</td>
<td>i &gt; count</td>
<td>i ( \neq 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]

### 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.
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 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

```plaintext
(array_up)
capacity

Push x on stack representation:
representation[count] := x

count := count + 1
```
A stack, concrete object

```
(count)
representation
```

“Push” x on stack representation:
representation [count] := x
count := count + 1

```
(free)
representation
```

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

```
A stack, concrete object
```

```
Stack: An Abstract Data Type (ADT)
```

## Types:

```
STACK [G]
```

\( \Rightarrow \): 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
```

```
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
    
    \[
    \text{inverse} (x) = 1 / x
    \]
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.

\[
\text{expression reduction value } = \text{ item } \left( \text{ remove } \left( \text{ put } \left( \text{ put } \left( \text{ put } \left( \text{ new } \right), \text{x8} \right), \text{x7} \right), \text{x6} \right) \right) \right)
\]
Expression reduction

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

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

**Expressed differently**

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

**An operational view of the expression**

**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: $OTHER \rightarrow T$ e.g. `new`
  - Queries: $T \times ... \rightarrow OTHER$ e.g. `item, empty`
  - Commands: $T \times ... \rightarrow T$ 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:
  - $STACK \ [G] \rightarrow G$: Formal generic parameter
- Functions (Operations):
  - `put`: $STACK \ [G] \times G \rightarrow STACK \ [G]$
  - `remove`: $STACK \ [G] \rightarrow STACK \ [G]$
  - `item`: $STACK \ [G] \rightarrow G$
  - `empty`: $STACK \ [G] \rightarrow BOOLEAN$
  - `new`: $STACK \ [G]$

### 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

**Secret part:**
- Choice of representation ($E2$)
- Implementation of functions by features ($E3$)

**Public part:**
- ADT specification ($E1$)
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

A very deferred class

defered class
  COUNTER

  feature
    item: INTEGER is
      deferred
        up is -- Increase item by 1.
          item := old item + 1
        end

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

  invariant
    item >= 0
  end
End of lecture 2