Programming in the large

Bertrand Meyer

Agenda for today

- Software quality
- Modularity
- Reusability

Software quality

- **External** factors: visible to customers
  - (not just end users but e.g. purchasers)
  - *Examples*: ease of use, extendibility
- **Internal** factors: perceptible only to developers
  - *Examples*: good programming style, information hiding

Only external factors count in the end, but the internal factors make it possible to obtain them.

External quality factors

- **CORRECTNESS**
- **ROBUSTNESS**
- **INTEGRITY**
- **EASE OF USE**
- **REUSABILITY**
- **EXTENDIBILITY**
- **PORTABILITY**
- **EFFICIENCY**
- ...

- Correctness:
  - The ability of a software system to perform according to specification, in cases defined by the specification.
- Robustness:
  - The ability of a software system to react in a reasonable manner to cases not covered by the specification.

Reliability

- Correctness + Robustness

- Techniques will be studied in detail: typing, Design by Contract, ...
Modularity

- Reusability + Extendibility
- Favored by architectural techniques tending to ensure decentralization of modules

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

Sequence

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


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

balance = list_of_deposits.total – list_of_withdrawals.total

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

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.
An object has an interface

An object has an implementation

Information hiding

The Open-Closed principle

Closing modules prematurely

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.)

```plaintext
 type PUBLICATION =
 record
   author, title: STRING;
   publication_year: INTEGER
   case pubtype: (book, journal, conference) of
     book: (publisher: STRING);
     journal: (editor: STRING);
     conference: (place, chair: STRING)
   end
 end
```

Use in clients:

```plaintext
 p: PUBLICATION;
 case p.pubtype of
   book: ... p.publisher ...;
   journal: ... p.editor ...;
   conference: ... p.place ...;
 end
```
Reusability issues

- Organizational and managerial issues:
  - (Not covered here.)
- Technical issues: what form of components?
  - Routine libraries
  - Packages (Ada)
  - Class libraries
  - What form of classes?

Reusability: Technical issues

- The general pattern for a searching routine:

  ```pascal
  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
  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?

  ```pascal
  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:

    ```pascal
    has (...) : 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
Factoring out commonality

- **TABLE**
  - start
  - after
  - found

Implementation variants

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>Linked list</td>
<td>c := first cell</td>
<td>c := c.right</td>
<td>c := Void</td>
</tr>
<tr>
<td>File</td>
<td>rewind</td>
<td>read</td>
<td>end_of_file</td>
</tr>
</tbody>
</table>

Encapsulation languages ("Object-based")

- Ada, Modula-2, 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.

Complementary material

- **OOSC2:**
  - Chapter 3: Modularity
  - Chapter 4: Approaches to reusability

End of lecture 2