Course objectives

- Provide you with solid knowledge of:
  - Object technology principles and methods
  - The practice of object-oriented analysis, design and implementation
  - Some open issues
  - Some recent developments
  - A specific technology

Contact

- Chair of Software Engineering:
  - http://se.inf.ethz.ch

- Course page:
  - http://se.inf.ethz.ch/teaching/ss2005/0250/

- Course assistant:
  - Ilinca Ciupa
  - http://se.inf.ethz.ch/people/ciupa

Grading

- 60% project
- 40% exam (last lecture day of semester)

Note: no out-of-semester exam
The project

- Theme
  - An Object Spyglass

- Documentation
  - User guide: how to use the tool
  - Developer guide: description of the architecture, main classes, limitations, how to extend the tool

- Test suite
  - Thorough set of test cases

Grading criteria

- Design (30 points)
  - Soundness (5 points)
  - Extendibility (5 points)
  - Ease of use (5 points)
  - Minimal requirements (15 points)

- Quality of contracts (20 points)
  - Documentation (20 points)
  - User guide (10 points)
  - Developer guide (10 points)

- Test (10 points)
  - Quality of test suite (5 points)
  - Correctness of the tool (5 points)

- Quality of code (10 points)
  - Style guidelines (5 points)
  - Quality of code (5 points)

- Effort devoted to the project (10 points)

Some words of warning

- Steps in reacting to O-O (from the preface to Object-Oriented Software Construction):
  - "(1) It’s trivial;
  - (2) It’s wrong;
  - (3) That’s how I did it all along anyway."

  Beware of the “mOOzak” phenomenon.

Reading assignment for next week

- OOSC, chapters
  - 3: Modularity
  - 6: Abstract data types
Some words of warning (cont’d)

benefit_from_course is
    -- Make students succeed.
    require
        some_humility
    do
        all_exercises
    ensure
        OO_mastery_for_fun_and_profit
end

Software engineering

The collection of processes, methods, techniques, tools and languages for developing quality operational software.

Terminology

- I will be strict about terminology:
  - Endless confusions in the literature and in discussions.
  - Basic concepts have precise definitions — no justification whatsoever for such confusions.
  - Object technology is (in part) about bringing rational, scientific principles to software. No excuse for sloppy terminology.

- Alternative conventions will be mentioned when necessary.

- CHF 5 fine for saying “object” when meaning “class” (after lecture 4)

External quality factors

Product quality (immediate):
- Correctness
- Robustness
- Security
- Ease of use
- Ease of learning
- Efficiency

Product quality (long-term):
- Extendibility
- Reusability
- Portability

Process quality:
- Timeliness
- Cost-effectiveness
Reliability

Correctness:
The systems’ ability to perform according to specification, in cases covered by the specification

Robustness:
The systems’ ability to perform reasonably in cases not covered by the specification

Security (integrity):
The systems’ ability to protect itself against hostile use

Modularity

- Reusability + Extendibility

- Favored by architectural techniques tending to ensure decentralization of modules

Reliability

- Correctness + Robustness + Security

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

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 design**

  
  [Link](http://www.acm.org/classics/dec95/)

**Top-down functional design**

**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.

Small interfaces principle

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

Few interfaces principle

- Every module communicates with as few others 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: An example**

$$balance = list\_of\_deposits\.total - list\_of\_withdrawals\.total$$

**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.

Information hiding

- Justifications:
  - Continuity
  - Decomposability

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, baseline 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**

Information hiding

An object has an **implementation**

The Open-Closed principle
Closing modules prematurely

```pascal
type PUBLICATION = record
  author, title: STRING;
  publication_year: INTEGER
end
```

- Use in clients:

```pascal
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?

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

- 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, ...

```java
// Example routine text for has
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`:

    ```java
    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
    ```

- **Representation independence:**
  - Can a client request an operation such as table search (`has`) without knowing what implementation is used internally?

```
has(t1, y)
```
Factoring out commonality

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.

Implementation variants

<table>
<thead>
<tr>
<th>start</th>
<th>forth</th>
<th>after</th>
<th>found (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>forth</td>
<td>after</td>
<td>found (x)</td>
</tr>
<tr>
<td>Array</td>
<td>$i := 1$</td>
<td>$i := i + 1$</td>
<td>$i &gt; \text{count}$</td>
</tr>
<tr>
<td>Linked list</td>
<td>$c := \text{first cell}$</td>
<td>$c := c.\text{right}$</td>
<td>$c = \text{Void}$</td>
</tr>
<tr>
<td>File</td>
<td>rewind</td>
<td>read</td>
<td>end_of_file</td>
</tr>
</tbody>
</table>

Complementary material

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