Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Lecture 9: Abstraction
Topics for today

Abstraction, especially functional abstraction

The notion of routine

The final word on features: all feature categories

The Uniform Access principle

Abstraction and client privileges

Information hiding
Routine: algorithm abstraction

To **abstract** is to capture the essence behind the details and the specifics.

Implies giving a *name* to the result.

In programming:

- Data abstraction: class
- Algorithm (operational) abstraction: routine
A routine is one of the two kinds of feature...

... the other is *attribute*

*We have encountered lots of routines already, without the name.*
A routine

\( r(\text{arg}: \text{TYPE}; ... ) \) is

-- Header comment

require

Precondition \((\text{boolean expression})\)

\( \text{do} \)

Body \((\text{instructions})\)

ensure

Postcondition \((\text{boolean expression})\)

end
Uses of routines

Bottom-up: capture existing algorithm, possibly for reuse

Top-down: **placeholder routines** — attractive alternative to pseudocode.

- **build_a_line**
  ```
  build_a_line is
  -- Build imaginary line.
  do
    Paris.display
    Metro.highlight
  create_fancy_line
  end
  ```

- **create_fancy_line**
  ```
  create_fancy_line is
  -- Create line and fill
  -- stations.
  do
    -- To be completed
  create_fancy_line
  end
  ```
Two kinds of routine

Procedure: doesn’t return a result

- Yields a command
- Calls are instructions

Function: returns a result

\[ f(\text{arg}: \text{TYPE}; \ldots): \text{RESULT\_TYPE} \text{ is} \]
\[ \ldots (\text{The rest as before}) \ldots \]

- Yields a query
- Calls are expressions
Features: the full story

A class is characterized by its features
Each feature is an operation on the corresponding objects:
query or command

Features are grouped into categories for readability

Class clauses:
- Indexing
- Inheritance
- Creation
- Feature (any number)
- Invariant

Anatomy of a class:
Features: the full story

Client view (specification)

Command

Response

No result

Internal view (implementation)

Routine

Computation

Memory

Function

Computation

Memory

Attribute

Feature

Query

Returns result
Uniform access principle

It doesn't matter to the client whether you look up or compute

A call such as

your_account.balance

could use an attribute or a function
Uniform Access: an example

\[ \text{balance} = \text{list}_\text{of}_\text{deposits}.\text{total} - \text{list}_\text{of}_\text{withdrawals}.\text{total} \]
Uniform Access Principle

Expressed more technically:

Features should be accessible to clients the same way whether implemented by storage or by computation.
An object has an **interface**
An object has an \textbf{implementation}
Information hiding
What clients may do

class METRO_STATION feature

\[ x, \ y: \text{REAL} \]
\hspace{1em} -- Coordinates of metro station

\[ \text{size}: \text{REAL} \]
\hspace{1em} -- Size of bounding square

\[ \text{upper}_\text{left}: \text{POSITION} \]
\hspace{1em} -- Upper-left position of bounding square

\[ \text{adjust\_positions \ is} \]
\hspace{1em} -- Set positions of bounding square

\[ \text{do} \]
\hspace{2em} \[ \text{upper\_left} . \text{set} (x - \text{size}/2, \ y + \text{size}/2) \]

\[ \text{end} \]
\hspace{1em} ...

\[ \text{end} \]
What clients may **not do**

```haskell
class METRO_STATION feature

  x, y: REAL
    -- Coordinates of metro station
  size: REAL
    -- Size of bounding square

  upper_left: POSITION
    -- Upper-left position of bounding square

  adjust_positions is
    do
      upper_left.x := 3
      ...
    end

end
```

**NOT PERMITTED!**
Use procedures:

```
upper_left.set(3, upper_left.y)
upper_left.set_x(3)
upper_left.move(3, h)
```
Possible client privileges

If class \( A \) has an attribute \( att : \text{SOME\_TYPE} \), what may a client class \( C \) with
\[
a : A
\]
do with \( a \cdot att \) ?

The attribute may be:

- Secret
- Read-only
- Read, restricted write
- Full write

Example: modify \( x \) with \( move \) in \( POINT \)

Modify through “set_...” procedure
Possible client privileges

If class $A$ has an attribute $att : SOME\_TYPE$, what may a client class $C$ with $a : A$ do with $a \cdot att$?

The attribute may be:

- Secret
- Read-only
- Read, restricted write
- Full write

Modify through $a \cdot some\_procedure$

Modify through $a \cdot set\_att(v)$

$a \cdot att$ invalid

$a \cdot att$ permitted in $C$ (for access)
If class $A$ has an attribute $\textit{att} : \textit{SOME\_TYPE}$, what may a client class $C$ with $a : A$ do with $a.\textit{att}$?

**Read access if attribute is exported**

- $a.\textit{att}$ is an expression.

- An assignment $a.\textit{att} := v$ would be syntactically illegal!

(It would assign to an expression, like $x + y := v$.)
Applying abstraction principles

Beyond read access: full or restricted write, through exported procedures.

Full write privileges: \textit{set\_attribute} procedure, e.g.

\begin{verbatim}
set_temperature (u: REAL) is
  -- Set temperature value to u.
  do
    temperature := u
  ensure
    temperature_set: temperature = u
  end
\end{verbatim}

Client will use e.g. \texttt{x.set\_temperature (21.5)}.
Other uses of a setter procedure

\begin{verbatim}
set_temperature (u : REAL) is
    -- Set temperature value to u.
    require
        not_under_minimum: u >= -273
        not_above_maximum: u <= 2000
    do
        temperature := u
        update_database
    ensure
        temperature_set: temperature = u
    end
\end{verbatim}
Having it both ways

Make it possible to call a setter procedure

*temperature: REAL* assign *set_temperature*

Then the syntax

```
x.temperature := 21.5
```

is accepted as a shorthand for *x.set_temperature (21.5)*

Retains contracts etc.
Information hiding

```
class A

feature
  f ...
  g ...

feature {NONE}
  h, i ...

feature {B, C}
  j, k, l ...

feature {A, B, C}
  m, n...
end

Status of calls in a client with a1: A:

- **a1.f, a1.g**: valid in any client
- **a1.h**: invalid everywhere
  (including in A's own text!)
- **a1.j**: valid only in B, C and their descendants
  (not valid in A!)
- **a1.m**: valid in B, C and their descendants, as well as in A and its descendants
```
An example of selective export

**LINKABLE** exports its features to **LINKED_LIST**

- Does not export them to the rest of the world
- Clients of **LINKED_LIST** don’t need to know about **LINKABLE** cells.
Exporting selectively

class LINKABLE[\text{G}]

feature \{\text{LINKED_LIST}\}

\begin{align*}
\text{put_right(...)} & \text{ is do ... end} \\
\text{right: G} & \text{ is do ... end} \\
& \text{...} \\
& \text{end}
\end{align*}

These features are selectively exported to \text{LINKED_LIST} and its descendants (and no other classes)
Information hiding

Information hiding only applies to use by clients, using dot notation or infix notation, as with \texttt{a1.f} (\textbf{Qualified} calls).

\textbf{Unqualified} calls (within class) not subject to information hiding:

```plaintext

class A feature {NONE}
   h is ... do ... end

feature

   f is
      do ...
      ...; h; ...
   end

end
```
What we have seen

The full categorization of features
Routines, procedures, functions
Uniform access
Information hiding
Selective exports
Setters and getters
Eiffel: assigner commands