Lecture 11: 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

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

\begin{itemize}
  \item \textbf{require}
    \begin{itemize}
      \item \textit{Precondition} (boolean expression)
    \end{itemize}
  \end{itemize}

\begin{itemize}
  \item \textbf{do}
    \begin{itemize}
      \item \textit{Body} (instructions)
    \end{itemize}
  \end{itemize}

\begin{itemize}
  \item \textbf{ensure}
    \begin{itemize}
      \item \textit{Postcondition} (boolean expression)
    \end{itemize}
  \end{itemize}

\textbf{end}
Uses of routines

Bottom-up: capture existing algorithm, possibly for reuse

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

\[
\text{build}_\text{a}_\text{line} \text{ is} \\
\quad -- \text{Build imaginary line} \\
\quad \text{do} \\
\quad\quad \text{Paris}.\text{display} \\
\quad\quad \text{Metro}.\text{highlight} \\
\quad\quad \text{create}_\text{fancy}_\text{line} \\
\quad \text{end}
\]

\[
\text{create}_\text{fancy}_\text{line} \text{ is} \\
\quad -- \text{Create line and fill stations} \\
\quad \text{do} \\
\quad\quad -- \text{To be completed} \\
\quad\quad -- \text{BM, 30 Oct 07} \\
\quad \text{end}
\]
Two kinds of routine

**Procedure**: doesn’t return a result
- Yields a **command**
- **Calls are instructions**

**Function**: returns a result

\[
f(\text{arg: TYPE; } ...): \text{RESULT\_TYPE} \text{ is}
\]
\[
... (\text{The rest as before}) ...
\]
- 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: Demo
Features: the full story

Client view
(specification)

Command → Procedure → Routine

Feature → No result → Routine

Feature → Returns result → Function

Query → Computation

Computation → Memory

Attribute

Internal view
(implementation)
Uniform access principle

A call such as

\texttt{your\_account\_balance}

could use an attribute or a function

It doesn't matter to the client whether you look up or compute
Uniform Access: an example

\[ balance = \text{list\_of\_deposits.total} - \text{list\_of\_withdrawals.total} \]
An object has an interface
An object has an implementation.
Information hiding

set
set_x
set_y

Intro. to Programming, lecture 11: Abstraction
Uniform Access Principle

Expressed more technically:

Features should be accessible to clients the same way whether implemented by storage or by computation.
Uniform Access: an example

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

(A1) 

<table>
<thead>
<tr>
<th>list_of_deposits</th>
<th>list_of_withdrawals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(A2) 

<table>
<thead>
<tr>
<th>list_of_deposits</th>
<th>list_of_withdrawals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Uniform Access Principle

Features should be accessible to clients the same way whether implemented by storage or by computation
What clients may do

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  
  -- Set positions of bounding square
    do
      upper_left . set (x - size/2, y + size/2)
    end

  ...

end

end
What clients may not do

class METRO_STATION feature

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

end
Use procedures:

```python
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 $att : SOME_TYPE$, 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

- $a \cdot att$ invalid
- $a \cdot att$ permitted in $C$ (for access)

Modify through $a \cdot some\_procedure$ $a \cdot set\_att(v)$
Abstraction and client privileges

If class \( A \) has an attribute \( att : SOME\_TYPE \), what may a client class \( C \) with

\[
\begin{align*}
a & : A \\
\end{align*}
\]
do with \( a \cdot att \)?

Read access if attribute is exported

- \( a.att \) is an expression.
- An assignment \( a.att := v \) 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: `set_attribute` procedure, e.g.

```pascal
set_temperature (u: REAL) is
  -- Set temperature value to u.
  do
    temperature := u
  ensure
    temperature_set: temperature = u
  end
```

Client will use e.g. `x.set_temperature (21.5)`. 
Other uses of a setter procedure

\textit{set\_temperature}(u : \textit{REAL}) \textbf{is}

\hspace{10pt} -- Set \textit{temperature} value to \textit{u}.

\textbf{require}

\hspace{10pt} \textit{not\_under\_minimum}: u \geq -273

\hspace{10pt} \textit{not\_above\_maximum}: u \leq 2000

\textbf{do}

\hspace{10pt} \textit{temperature} := u

\hspace{10pt} \textit{update\_database}

\textbf{ensure}

\hspace{10pt} \textit{temperature\_set}: \textit{temperature} = u

\textbf{end}
Having it both ways

Make it possible to call a setter procedure

\[
\text{temperature: REAL } \text{assign set_temperature}
\]

Then the syntax

\[
x.\text{temperature} := 21.5
\]

is accepted as a shorthand for \(x.\text{set_temperature}(21.5)\)

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

\[
\begin{align*}
\text{class} & \quad \text{LINKABLE}[G] \\
\text{feature} & \quad \{\text{LINKED\_LIST}\} \\
\text{put\_right}(\ldots) & \quad \text{is do ... end} \\
\text{right. } G & \quad \text{is do ... end} \\
\ldots & \\
\text{end}
\end{align*}
\]

These features are selectively exported to \textit{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
      ...; \textcolor{red}{h}; ...
    end
end
```
What we have seen
What we have seen

- Routines, procedures, functions
- The full categorization of features
- More on information hiding
- Uniform access
- Selective exports
- Feature categories
- Setters and getters
- Eiffel: assigner commands
End of lecture 11