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* of a concept, ignoring details & specifics

Implies:
- *Removing* some information
- *Giving a name* to the result of the abstraction

In programming:
- Data abstraction: *class*
- Algorithm (operational) abstraction: *routine*

A routine is also known as a *method*

And also as a *subprogram* and as a *subroutine*
A routine is one of the two kinds of feature...

... the other is \textit{attribute}

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

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

\[ \text{-- Header comment.} \]

\textbf{require}

\[ \textit{Precondition} (\text{boolean expression}) \]

\textbf{do}

\[ \textit{Body} (\text{instructions}) \]

\textbf{ensure}

\[ \textit{Postcondition} (\text{boolean expression}) \]

\textbf{end}
Uses of routines

Bottom-up: capture existing algorithm, possibly for reuse

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

**Methodology:** "FIXME" entries should be informative

---

**build_a_line**

-- Build imaginary line.

do

*Paris.display*

*Metro.highlight*

create_fancy_line

end

**create_fancy_line**

-- Create line and fill

-- stations.

do

-- To be completed

-- BM, 26 Oct 09

end
Two kinds of routine

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

**Function**: returns a result

\[ f(\text{arg} : \text{TYPE}; \ldots) : \text{RESULT\_TYPE} \]

... (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:
Features: the full story

Client view (specification)

Command → Procedure

Feature

No result

Returns result

Query

Internal view (implementation)

Routine

Computation

Memory

Feature

Function

Attribute

Computation

Memory
The Uniform Access principle

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

\[ balance = list_{of\_deposits}.total - list_{of\_withdrawals}.total \]

(A1)

\begin{align*}
list_{of\_deposits} & \quad (A1) \\
list_{of\_withdrawals} & \\
balance & 
\end{align*}

(A2)

\begin{align*}
list_{of\_deposits} & \quad (A2) \\
list_{of\_withdrawals} & 
\end{align*}

A call such as 
\texttt{your\_account.balance}

could use an attribute or a function
The Uniform Access principle

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

![Diagram of an object with methods set_x and set_y]
Information hiding
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
    -- Set positions of bounding square.
    do
      upper_left.set(x - size/2, y + size/2)
    end

end
What clients may not 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  
    -- Set positions of bounding square.
      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)
```
Abstraction and client privileges

If class $A$ has an attribute $x$, what may a client class $C$ do with $a1.x$ for $a1$ of type $A$?

Read access if attribute is exported

- $a1.x$ is an expression!

- An assignment $a1.x := v$ would be syntactically illegal!

(It would assign to an expression, like $a+b := v$)
Applying abstraction principles

To provide clients with writing privileges: define a setter procedure, such as

\[
\text{set\_temperature}(u:\text{REAL})
\]

\[
\quad \text{-- Set temperature value to } u.
\]

\[
\begin{align*}
\text{do} & \\
\text{temperature} & := u \\
\text{ensure} & \\
\text{temperature\_set: temperature} & = u
\end{align*}
\]

Clients will use calls such as

\[
x.\text{set\_temperature}(21.5)
\]
Taking full advantage of a setter command

\textit{set\_temperature}(u : REAL)
\begin{itemize}
\item \textit{-- Set temperature value to } \textit{u}.
\end{itemize}
\begin{itemize}
\item \textbf{require}
\item \text{not\_under\_minimum}: \textit{u} \geq -273
\item \text{not\_above\_maximum}: \textit{u} \leq 2000
\end{itemize}
\begin{itemize}
\item \textbf{do}
\item \textit{temperature} := \textit{u}
\item \textbf{update\_database}
\end{itemize}
\begin{itemize}
\item \textbf{ensure}
\item \text{temperature\_set}: \textit{temperature} = \textit{u}
\end{itemize}
\begin{itemize}
\item \textbf{end}
\end{itemize}
Abstraction and client privileges

If class $A$ has an attribute $x$, what may a client class $C$ do with $a1.x$ for $a1$ of type $A$?

Read access if attribute is exported

- $a1.x$ is an expression!
- An assignment $a1.x := v$ would be syntactically illegal!

(It would assign to an expression, like $a + b := v$)
Exporting (making public) an attribute

In Eiffel, exporting an attribute means exporting it read-only.

From the outside, it is not shown as an attribute, just as a **query**: it could be a function.

In C++, Java & C#, if you make public an attribute* $x$, it is available for both read and write:

- $v := a1.x$
- $a1.x := v$

As a result, it is almost always a bad idea to export an attribute.

* (field, member variable)
Getter functions

In C++, Java & C#, the standard technique, if `private_x` is secret, is to export an associated getter function:

```
x: T
  do
    Result := private_x
  end
```

Eiffel needs no getter functions: just export the attribute.

This is safe: the attribute is exported

- Only for reading
- Without the information that it is an attribute: it could be a function (Uniform Access principle)
Having it both ways (Eiffel syntax)

It is possible to define a query as

```eiffel
temperature: REAL assign set_temperature
```

Then the syntax

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

is accepted as an abbreviation for

```eiffel
x.set_temperature(21.5)
```

Retains contracts and any other supplementary operations

C# has a notion of “property” which pursues the same goal
Information hiding

<table>
<thead>
<tr>
<th>class</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature</td>
<td>f ...</td>
</tr>
<tr>
<td></td>
<td>g ...</td>
</tr>
<tr>
<td>feature</td>
<td>{NONE}</td>
</tr>
<tr>
<td></td>
<td>h, i ...</td>
</tr>
<tr>
<td>feature</td>
<td>{B, C}</td>
</tr>
<tr>
<td></td>
<td>j, k, l ...</td>
</tr>
<tr>
<td>feature</td>
<td>{A, B, C}</td>
</tr>
<tr>
<td></td>
<td>m, n ...</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

Status of calls in a client with \texttt{a1: A}:

- \texttt{a1.f, a1.g}: valid in any client
- \texttt{a1.h}: invalid everywhere (including in A's own text!)
- \texttt{a1.j}: valid only in B, C and their descendants (not valid in A!)
- \texttt{a1.m}: valid in B, C and their descendants, as well as in A and its descendants
Information hiding only applies to use by clients, using dot notation or infix notation, as with \texttt{a1.f} (\texttt{Qualified} calls).

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

```plaintext
class A feature {NONE}
  h do ... end
feature
  f
  do ...; h; ...
  end
end
```
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.

![Diagram showing the selective export of features from LINKABLE to LINKED_LIST]

- `count`: 3
- `first_element`
- `active`
- `item` and `right` connections
Exporting selectively

class

LINKABLE [G]

feature {LINKED_LIST}

put_right (...) do ... end

right: G do ... end

...

end
class LINKABLE feature {LINKED_LIST}
    item : STRING
        -- Value in this cell
    right : LINKABLE
        -- Cell, if any, to which this one is chained
    put_right (other : like Current)
        -- Put other to the right of current cell.
        do
            right := other
        ensure
            chained : right = other
        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
Reading assignment for next week

Chapters on

- Syntax (11)
- Inheritance (16)