Topics for today

- Abstraction, especially functional abstraction
- The notion of routine
- The final word on features: all feature categories

Tying up a few loose ends from last lecture:
- Low-level control structures, flowcharts
- Levenshtein (edit) distance algorithm
- Undecidability of the halting problem

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(arg: TYPE; ...) is
  -- Header comment
  require
    ... Precondition (boolean expression) ...
  do
    ... Body (instructions) ...
  ensure
    Postcondition (boolean expression)
end```
Uses of routines

Bottom-up: capture existing algorithm, possibly for reuse
Top-down: placeholder routines — attractive alternative to pseudocode.

```plaintext
build_a_line is 
  Build imaginary line 
  do 
    Paris.display 
    Menu.highlight 
    create_fancy_line 
  end

create_fancy_line is 
  -- Create line and fill stations 
  do 
    -- To be completed 
    BM, 28 Nov 06 
  end
```

Two kinds of routine

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

Function: returns a result
- $f$ (arg: TYPE ...): RESULT_TYPE is
  ... (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:

Uniform access principle

It doesn’t matter to the client whether you look up or compute
A call such as

```plaintext
your_account.balance
```

could use an attribute or a function

Uniform Access: an example

```plaintext
balance = list_of_deposits.total - list_of_withdrawals.total
```

(A1)

(A2)
An object has an interface

An object has an implementation

Information hiding

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

Uniform Access Principle

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

```java
class METRO_STATION
    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 begin
                upper_left.set(x - size/2, y - size/2)
            end
end
```

What clients may not do

```java
class METRO_STATION
    adjust_positions
        -- Set positions of bounding square
            do begin
                upper_left.x := 3
            end
end
```

Use procedures:

```java
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.att?

The attribute may be:

- Secret
- Read-only
- Restricted write
- Full write
- Modify through 'set...

Example: modify x with move in POINT

Abstraction and client privileges

If class A has an attribute att: SOME_TYPE, what may a client class C with
- a: A
  do with a.att?

Read access if attribute is exported
> a.att is an expression.
> An assignment a.att := v would be syntactically illegal!

(a 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.

```plaintext
set_temperature (u: REAL) is
  do
    temperature := u
  end
ensure
  temperature_set: temperature = u
end
```

Client will use e.g. `x.set_temperature(21.5).`

Other uses of a setter procedure

```plaintext
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
  end
ensure
  temperature_set: temperature = u
end
```

Having it both ways

Make it possible to call a setter procedure

```plaintext
temperature: REAL assign set_temperature
```

Then the syntax

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

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

Retains contracts etc.

Information hiding

```plaintext
class A
  feature
    f...
    g...
  feature (NONE)
    h...
  feature (R, G)
    j, k...
  feature (A, R, G)
    m, n...
end
```

Status of calls in a client with `at A`:  

- `a1, a2, a3` valid in any client
- `a4` invalid everywhere (including in `A`'s own text)
- `a5` valid only in `R` and their descendants (not valid in `A`)
- `a6` valid in `R` and their descendants, as well as in `A` and its descendants

An example of selective export

```plaintext
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

```plaintext
class
  LINKABLE(G)
  feature (LINKED_LIST)
    put_right(...) is do ... end
    right: G is do ... end
    ...
end
```

These features are selectively exported to 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 $a.f$ (Qualified calls).

Unqualified calls (within class) not subject to information hiding:

```java
class A feature (NONE)
    h is ... do ... end
    ... feature
    is ... do ... end
    ... end
```

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