Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 6
Today

- Conditional
- Loop
- Abstractions
- Exporting features
Structured programming

- In structured programming instructions can be combined only in three ways (constructs):

  - **Compound**
    - Sequential composition
  - **Conditional**
    - True: $s_1$
    - False: $s_2$
  - **Loop**
    - $c$
      - True: $s_1$
      - False: $s_2$

- Each of these blocks has a single entry and exit and is itself a (possibly empty) compound
Conditional

- Basic syntax:
  ```
  if c then
  s_1
  else
  s_2
  end
  ```

- `c` is a boolean expression (e.g., entity, query call of type `BOOLEAN`)

- `else`-part is optional:
  ```
  if c then
    s_1
  end
  ```
Calculating function’s value

\[ f(\text{max: INTEGER; } s: \text{ STRING}): \text{ STRING} \]

do
  if \( s.\text{is}\_\text{equal}(\"Java\") \text{ then} \]
    Result := \"J**a\"
  else
    if \( s.\text{count} > \text{max} \text{ then} \]
      Result := \"<an unreadable German word>\"
    end
  end
end

Calculate the value of:

- \( f(3, \text{\"Java\")} \rightarrow \text{\"J**a\"} \)
- \( f(20, \text{\"Immatrikulationsbestätigung\")} \rightarrow \text{\"<an unreadable German word>\"} \)
- \( f(6, \text{\"Eiffel\")} \rightarrow \text{Void} \)

Write a routine...

- ... that computes the maximum of two integers:
  \[ \text{max}(a, b: \text{INTEGER}): \text{INTEGER} \]

- ... that increases time by one second inside class \textit{TIME}:

  \begin{verbatim}
  class \textit{TIME}
    hour, minute, second: \text{INTEGER}
    second_forth
      do ... end
    ...
  end
  \end{verbatim}
Comb-like conditional

If there are more than two alternatives, you can use the syntax:

```
if \( c_1 \) then
   \( s_1 \)
elseif \( c_2 \) then
   \( s_2 \)
...
elseif \( c_n \) then
   \( s_n \)
else
   \( s_e \)
end
```

instead of:

```
if \( c_1 \) then
   \( s_1 \)
else
   if \( c_2 \) then
      \( s_2 \)
   else
      ...
if \( c_n \) then
   \( s_n \)
else
   \( s_e \)
end
...
end
```
If all the conditions have a specific structure, you can use the syntax:

\[
\text{inspect expression when } \text{const}_1 \text{ then } s_1
\]
\[
\text{when } \text{const}_2 \text{ then } s_2
\]
\[
...\]
\[
\text{when } \text{const}_{n1} .. \text{const}_{n2} \text{ then } s_n
\]
\[
\text{else } s_e
\]
end

Integer or character expression

Integer or character constant

Compound

Interval
Lost in conditions

Rewrite the following multiple choice:

- using a comb-like conditional
- using nested conditionals

```plaintext
inspect user_choice
when 0 then
   print ("Hamburger")
when 1 then
   print ("Coke")
else
   print ("Not on the menu!")
end
```

```plaintext
if user_choice = 0 then
   print ("Hamburger")
elseif user_choice = 1 then
   print ("Coke")
else
   print ("Not on the menu!")
end
```

```plaintext
if user_choice = 0 then
   print ("Hamburger")
else
   if user_choice = 1 then
      print ("Coke")
   else
      print ("Not on the menu!")
   end
else
   print ("Hamburger")
end
```
Loop: Basic form

Syntax:

```
from initialization Compound
until exit_condition Boolean expression
loop body Compound
end
```
Compilation error? Runtime error?

\[ f(x, y: \text{INTEGER}): \text{INTEGER} \]

do
  from 
  until \((x \div y)\)
  loop
    "Print me!"
  end
end

Compilation error: integer expression instead of boolean

Compilation error: expression instead of instruction

Correct

\[ f(x, y: \text{INTEGER}): \text{INTEGER} \]

local
  \(i: \text{INTEGER}\)
  do
    from \(i := 1\)
    until (True)
    loop
      \(i := i \times x \times y\)
    end
  end
Simple loop

How many times will the body of the following loop be executed?

```eiffel
i: INTEGER

... from i := 1 until i > 10 loop
    print ("I will not say bad things about assistants")
    i := i + 1
end

... from i := 10 until i < 1 loop
    print ("I will not say bad things about assistants")
end
```

Caution! Loops can be infinite!
What does this function do?

\[
\text{factorial} (n: \text{INTEGER}): \text{INTEGER} \\
\text{require} \\
\quad n \geq 0 \\
\text{local} \\
\quad i: \text{INTEGER} \\
\text{do} \\
\quad \text{from} \\
\quad \quad i := 2 \\
\quad \quad \text{Result} := 1 \\
\quad \text{until} \\
\quad \quad i > n \\
\quad \text{loop} \\
\quad \quad \text{Result} := \text{Result} \times i \\
\quad \quad i := i + 1 \\
\quad \text{end} \\
\text{end}
\]
Loop: More general form

Syntax:

```
from initialization invariant inv until exit_condition loop body variant var end
```

- `from` and `until` are required.
- `invariant` (with `inv`) is optional.
- `variant` (with `var`) is optional.
- `initialization` and `exit_condition` are optional.
- `Boolean expression` is required.
- `Compound expression` is required.
- `Integer expression` is optional.
Invariant and variant

Loop invariant (do not confuse with class invariant)
- holds before and after the execution of loop body
- captures how the loop iteratively solves the problem: e.g. “to calculate the sum of all \( n \) elements in a list, on each iteration \( i (i = 1..n) \) the sum of first \( i \) elements is obtained”

Loop variant
- integer expression that is nonnegative after execution of from clause and after each execution of loop clause and strictly decreases with each iteration
- a loop with a correct variant can not be infinite (why?)
What are the invariant and variant of the “factorial” loop?

from

\[ i := 2 \]
\[ \text{Result} := 1 \]

invariant

\[ \text{Result} = \text{factorial} (i - 1) \]

until

\[ i > n \]

loop

\[ \text{Result} := \text{Result} \times i \]
\[ i := i + 1 \]

variant

\[ n - i + 2 \]

end
Implement a function that calculates Fibonacci numbers, using a loop

\[ \text{fibonacci}(n: \text{INTEGER}): \text{INTEGER} \]

\[ -- \text{n-th Fibonacci number} \]

\[ \text{require} \]

\[ n\_\text{non\_negative}: n \geq 0 \]

\[ \text{ensure} \]

\[ \text{first\_is\_zero}: n = 0 \implies \text{Result} = 0 \]

\[ \text{second\_is\_one}: n = 1 \implies \text{Result} = 1 \]

\[ \text{other\_correct}: n > 1 \implies \text{Result} = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2) \]

\[ \text{end} \]
Writing loops (solution)

\[
\text{fibonacci}(n: \text{INTEGER}): \text{INTEGER}
\]

local
\[
a, b, i: \text{INTEGER}
\]
do
\[
\text{if } n \leq 1 \text{ then}
\]
\[
\text{Result} := n
\]
else
\[
\text{from}
\]
\[
a := \text{fibonacci}(0)
\]
\[
b := \text{fibonacci}(1)
\]
\[
i := 1
\]
\[
\text{invariant}
\]
\[
a = \text{fibonacci}(i - 1)
\]
\[
b = \text{fibonacci}(i)
\]
until
\[
i = n
\]
loop
\[
\text{Result} := a + b
\]
\[
a := b
\]
\[
b := \text{Result}
\]
\[
i := i + 1
\]
\[
\text{variant}
\]
\[
n - i
\]
end
end
Abstraction

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

The client is interested in:

- a **set of services** that a software module provides, not its internal **representation**
  - hence, the class abstraction
- what a service does, not how it does it
  - hence, the feature abstraction
- Programming is all about finding right abstractions
- However, the abstractions we choose can sometimes fail, and we need to find new, more suitable ones.
Finding the right abstractions (classes)

Suppose you want to model your room:

```ruby
class ROOM
  feature
    -- to be determined
  end

Your room probably has thousands of properties and hundreds of things in it.

Therefore, we need a first abstraction: What do we want to model?

In this case, we focus on the size, the door, the computer and the bed.
Finding the right abstractions (classes)

To model the size, an attribute of type `DOUBLE` is probably enough, since all we are interested in is its value:

class *ROOM*

feature

    size: DOUBLE

    -- Size of the room.

end
Finding the right abstractions (classes)

Now we want to model the door. If we are only interested in the state of the door, i.e. if it is open or closed, a simple attribute of type BOOLEAN will do:

class ROOM

feature

  size: DOUBLE
    -- Size of the room.

  is_door_open: BOOLEAN
    -- Is the door open or closed?

end
Finding the right abstractions (classes)

But what if we are also interested in what our door looks like, or if opening the door triggers some behavior?

- Is there a daring poster on the door?
- Does the door squeak while being opened or closed?
- Is it locked?
- When the door is being opened, a message will be sent to my cell phone

In this case, it is better to model a door as a separate class!
Finding the right abstractions (classes)

class ROOM
feature
    size: DOUBLE
        -- Size of the room
        -- in square meters.
    door: DOOR
        -- The room’s door.
end

class DOOR
feature
    is_locked: BOOLEAN
        -- Is the door locked?
    is_open: BOOLEAN
        -- Is the door open?
    is_squeaking: BOOLEAN
        -- Is the door squeaking?
    has_daring_poster: BOOLEAN
        -- Is there a daring poster on
        -- the door?
    open
        -- Opens the door
        do
            -- Implementation of open,
            -- including sending a message
        end
    -- more features...
end
Finding the right abstractions (classes)

How would you model...

... the computer?

... the bed?

How would you model an elevator in a building?
Finding the right abstractions (features)

(BANK_ACCOUNT)

```
| deposits | 1000 | 300 |
| withdrawals | 500 |
| balance | 800 |
```

**Invariant:** \( \text{balance} = \text{total (deposits)} - \text{total (withdrawals)} \)

Which one would you choose and why?
Exporting features: The stolen exam

class PROFESSOR

create

  make

feature

  make (a_exam_draft: STRING)
    do
      exam_draft := a_exam_draft
    end

feature

  exam_draft: STRING

end
class ASSISTANT

create
  make
feature
  make (a_prof: PROFESSOR)
    do
      prof := a_prof
    end
feature
  prof: PROFESSOR
feature
  review_draft
    do
      -- review prof.exam_draft
    end
end
Exploiting a hole in information hiding

class STUDENT

create
  make
feature
  make (a_prof: PROFESSOR; a_assi: ASSISTANT)
    do
      prof := a_prof
      assi := a_assi
    end
feature
  prof: PROFESSOR
  assi: ASSISTANT
feature
  stolen_exam: STRING
    do
      Result := prof.exam_draft
    end
end
Don’t try this at home!

you: STUDENT
your_prof: PROFESSOR
your_assi: ASSISTANT
stolen_exam: STRING

create your_prof.make ("top secret exam!"")
create your_assi.make (your_prof)
create you.make (your_prof, your_assistant)

stolen_exam := you.stolen_exam
Fixing the issue

class PROFESSOR
create
   make
feature
   make (a_exam_draft: STRING)
     do
       exam_draft := a_exam_draft
     end
feature {PROFESSOR, ASSISTANT}
   exam_draft: STRING
end
The export status does matter!

class STUDENT
create
  make
feature
  make (a_prof: PROFESSOR; a_assi: ASSISTANT)
    do
      prof := a_prof
      assi := a_assi
    end
feature
  prof: PROFESSOR
  assi: ASSISTANT
feature
  stolen_exam: STRING
    do
      Result := assi.prof.exam_draft
    end
end

Invalid call!
Exporting features

Status of calls in a client with $a1$ of type $A$:

- $a1.f$, $a1.g$: valid in any client

- $a1.h$: invalid everywhere (including in $A$'s text!)

- $a1.j$: valid in $B$, $C$ and their descendants (invalid in $A$!)

- $a1.m$: valid in $B$, $C$ and their descendants, as well as in $A$ and its descendants.
Compilation error?

class PERSON
feature
  name: STRING
feature {BANK}
  account: BANK_ACCOUNT
feature {NONE}
  loved_one: PERSON
  think
do
  print ("Thinking of " + loved_one.name)
end
lend_100_franks
do
  loved_one.account.transfer (account, 100)
end
Exporting attributes

Exporting an attribute only means giving read access

\[ x.f := 5 \]

Attributes of other objects can be changed only through commands

- protecting the invariant
- no need for getter functions!
Example

class TEMPERATURE
feature
  celsius_value: INTEGER

make_celsius (a_value: INTEGER)
  require
    above_absolute_zero: a_value >= - Celsius_zero
  do
    celsius_value := a_value
  ensure
    celsius_value_set := celsius_value = a_value
  end

end
Assigners

If you like the syntax

\[ x.f := 5 \]

you can declare an assigner for \( f \)

- In class \textit{TEMPERATURE}
  \[ \text{celsius\_value: INTEGER assign make\_celsius} \]
- In this case
  \[ t.\text{celsius\_value} := 36 \]
is a shortcut for
  \[ t.\text{make\_celsius} (36) \]
- ... and it won’t break the invariant!
Information hiding vs. creation routines

class **PROFESSOR**
create
    make
feature {None}
    make (a_exam_draft: STRING)
        do
            ...
        end
end
end

Can I create an object of type **PROFESSOR** as a client?

After creation, can I invoke feature **make** as a client?
Controlling the export status of creation routines

class PROFESSOR
create {COLLEGE_MANAGER}
  make
feature {None}
  make (a_exam_draft: STRING)
    do
      ...
    end
end

can I create an object of type PROFESSOR as a client? After creation, can I invoke feature make as a client? What if I have create {NONE} make instead of create {COLLEGE_MANAGER} make?