ETH Zürich
Chair of Software Engineering

Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

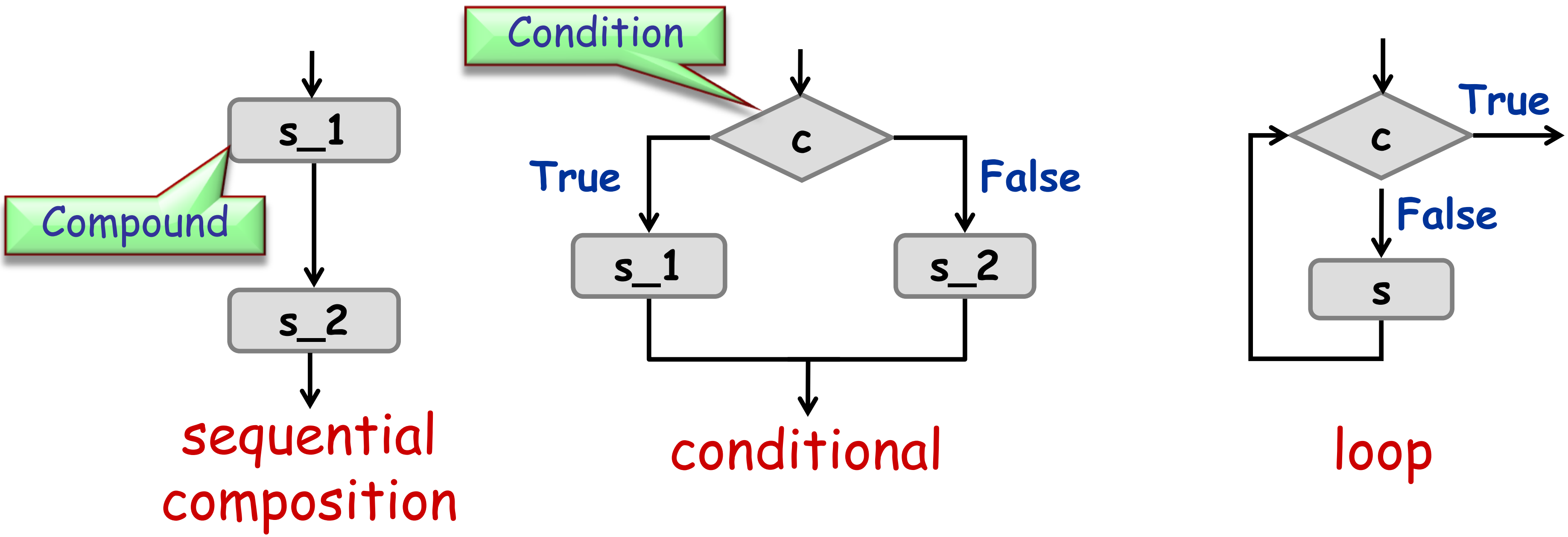
Exercise Session 6

Today

- Conditional
- Loop
- Abstractions
- Exporting features

Structured programming

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



➢ 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
```

Condition (c)
Compound (s_1)
Compound (s_2)

- *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

Hands-On

```
f (max: INTEGER; s: STRING): STRING
  do
    if s.is_equal ("Java") then
      Result := "J**a"
    else
      if s.count > max then
        Result := "<an unreadable German word>"
      end
    end
  end
```

Calculate the value of:

- *f* (3, "Java") → "J**a"
- *f* (20, "Immatrikulationsbestätigung") → "<an unreadable German word>"
- *f* (6, "Eiffel") → **Void**

Write a routine...

- ... that computes the maximum of two integers:

  max (a, b: INTEGER): INTEGER

- ... that increases time by one second inside class *TIME*:

```
class TIME
    hour, minute, second: INTEGER

    second_forth
        do ... end
    ...
end
```

Hands-On

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
```

Condition

Compound

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
end
```

Multiple choice

If all the conditions have a specific structure, you can use the syntax:

```
inspect expression
when const_1 then
    s_1
when const_2 then
    s_2
...
when const_n1 .. const_n2 then
    s_n
else
    s_e
end
```

Integer or character expression

Integer or character constant

Compound

Interval

Lost in conditions

Hands-On

Rewrite the following multiple choice:

- using a comb-like conditional
- using nested conditionals

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

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

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

Loop: Basic form

Syntax:

from

initialization — Compound

until

exit_condition — Boolean expression

loop

body — Compound

end

Compilation error? Runtime error?

Hands-On

```
f (x, y : INTEGER): INTEGER
    do
        from
        until (x // y)
        loop
            "Print me!"
        end
    end
```

Compilation error: integer expression instead of boolean

Compilation error: expression instead of instruction

```
f (x, y : INTEGER): INTEGER
    local
        i : INTEGER
    do
        from i := 1
        until (True)
        loop
            i := i * x * y
        end
    end
```

Correct

Simple loop

Hands-On

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

```
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
```

In Eiffel we usually start counting from 1

10

∞

Caution! Loops can be infinite!

What does this function do?

Hands-On

```
factorial (n: INTEGER): INTEGER
        require
            n >= 0
        local
            i: INTEGER
        do
            from
                i := 2
                Result := 1
            until
                i > n
            loop
                Result := Result * i
                i := i + 1
            end
        end
```

Loop: More general form

Syntax:

```
from
        initialization      -- Compound
invariant                   -- Optional
        inv                 -- Boolean expression
until
        exit_condition      -- Boolean expression
loop
        body                -- Compound
variant                     -- Optional
        var                 -- Integer expression
end
```

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?)

Invariant and variant

Hands-On

What are the invariant and variant of the "factorial" loop?

```
from
        i := 2
        Result := 1
invariant
        Result = factorial (i - 1)
until
        i > n
loop
        Result := Result * i
        i := i + 1
variant
        n - i + 2
end
```

Result = 6 = 3!

Writing loops

Hands-On

Implement a function that calculates Fibonacci numbers, using a loop

```
fibonacci (n: INTEGER): INTEGER
        -- n-th Fibonacci number
    require
        n_non_negative: n >= 0
    ensure
        first_is_zero: n = 0 implies Result = 0
        second_is_one: n = 1 implies Result = 1
        other_correct: n > 1 implies Result =
    fibonacci (n - 1) + fibonacci (n - 2)
    end
```

Writing loops (solution)

Hands-On

```
fibonacci (n: INTEGER): INTEGER
    local
        a, b, i: INTEGER
    do
        if n <= 1 then
            Result := n
        else
            from
                a := fibonacci (0)
                b := fibonacci (1)
                i := 1
            invariant
                a = fibonacci (i - 1)
                b = fibonacci (i )
            until
                i = n
            loop
                Result := a + b
                a := b
                b := Result
                i := i + 1
            variant
                n - i
            end
        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:

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

door material location bed desk computer size furniture etc shape etc etc messy?

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 it's 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)

Hands-On

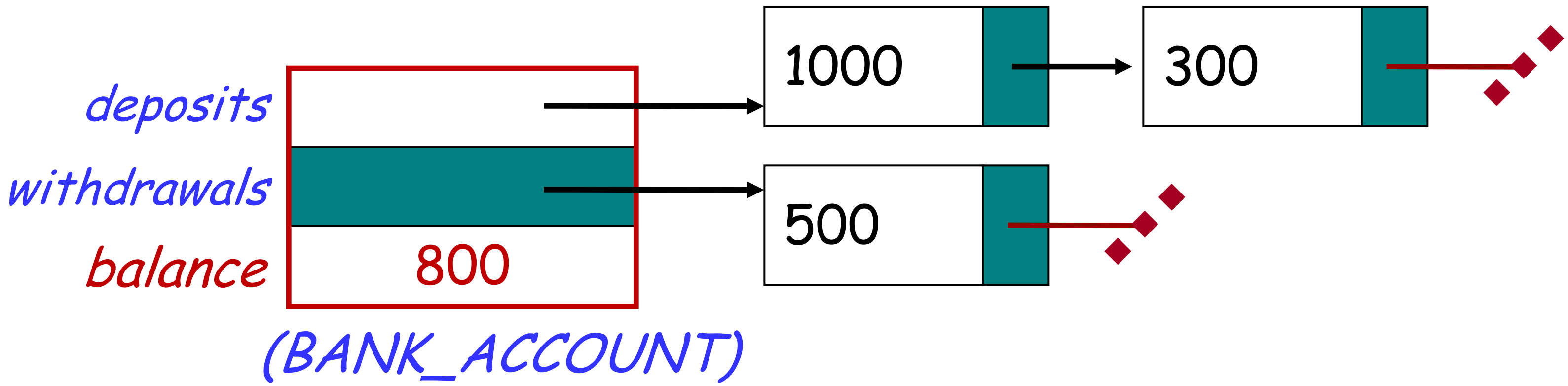
How would you model…

… the computer?

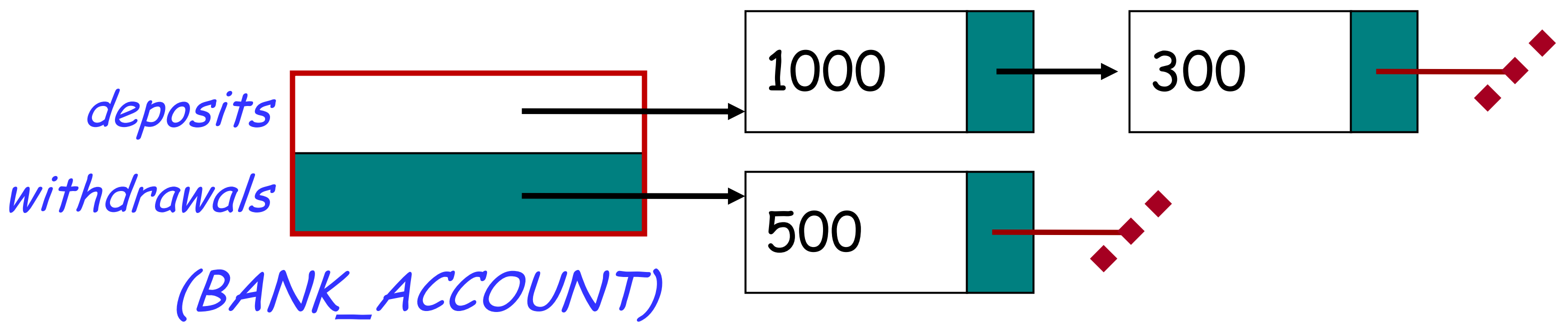
… the bed?

How would you model an elevator in a building?

Finding the right abstractions (features)



invariant: *balance = total (deposits) – 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
```

For your eyes only

```
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

Hands-On

```
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

```
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* 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?

Hands-On

```
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
end
```

OK: unqualified call

OK: exported to all

Error: not exported to PERSON

OK: unqualified call

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 *TEMPERATURE*

  celsius_value: INTEGER **assign** *make_celsius*

- In this case

t.celsius_value := 36

is a shortcut for

t.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
```

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* ?