



# Einführung in die Programmierung Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 7

# News (Reminder)

---



Mock exam next week! (November 10)

- You have to be present
- Assignment 7 due next Tuesday, as usual
- The week after we will discuss the results



- Abstractions
- Uniform Access Principle
- Naming conventions
- Exporting features



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**  
**class**
- **what** a service does, not **how** it does it  
**feature**
- Object-oriented programming is all about finding right abstractions
- However, the abstractions we choose can sometimes fail, and we need to find new, more suitable ones.



"That is, approximately, the magic of TCP. It is what computer scientists like to call an abstraction: a simplification of something much more complicated that is going on under the covers. As it turns out, a lot of computer programming consists of building abstractions. What is a string library? It's a way to pretend that computers can manipulate strings just as easily as they can manipulate numbers. What is a file system? It's a way to pretend that a hard drive isn't really a bunch of spinning magnetic platters that can store bits at certain locations, but rather a hierarchical system of folders-within-folders containing individual files that in turn consist of one or more strings of bytes."

(from <http://www.joelonsoftware.com/articles/LeakyAbstractions.html> )



What abstractions were used in the *temperature converter* from assignment 4?

- Why it is better to have a class for *TEMPERATURE* than to store the value in an *INTEGER* variable?
- How was the Celsius value obtained? What about the Kelvin value? Did you see that difference in the class *TEMPERATURE\_APPLICATION*?

# Finding the right abstractions (classes)

---



Suppose you want to model your room:

```
class ROOM
```

```
feature
```

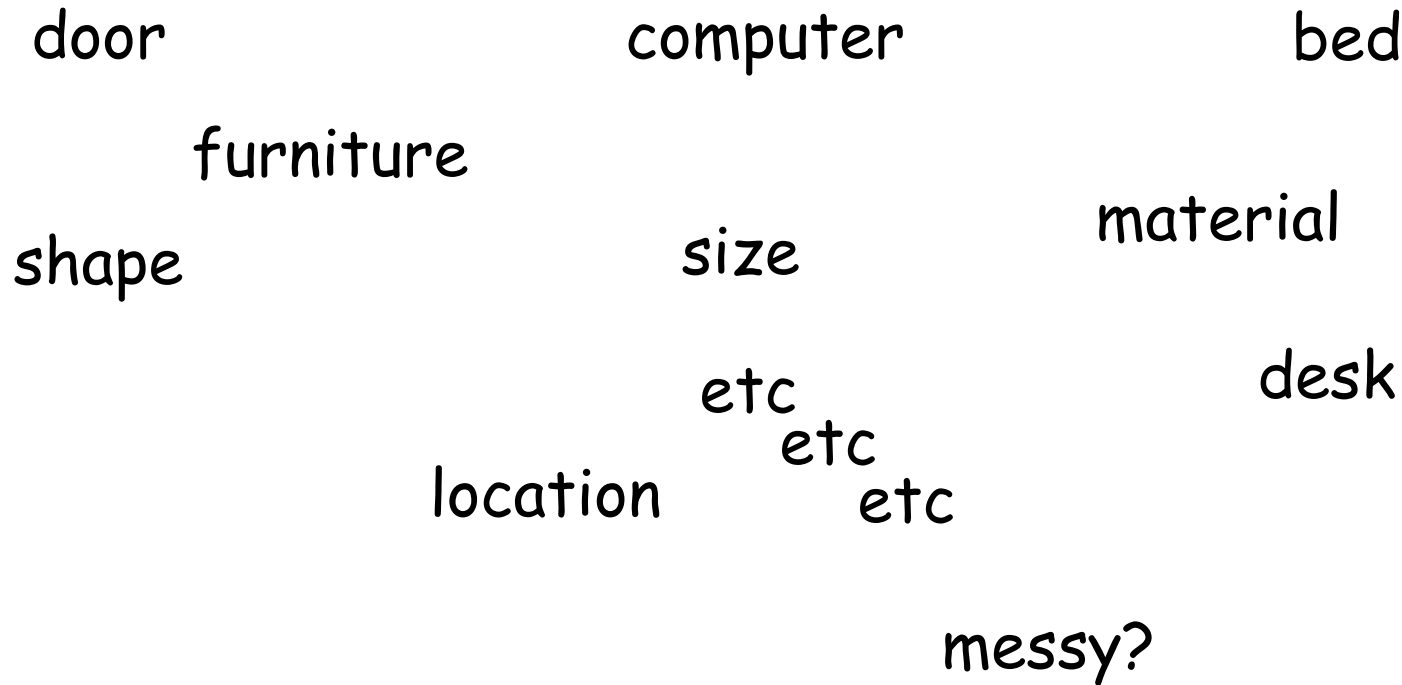
```
    -- to be determined
```

```
end
```

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

# Finding the right abstractions (classes)

---



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?

- Is there a poster on the door?
- Does it squeak when we close or open it?
- Is it locked?

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

# Finding the right abstractions (classes)



```
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_playboy_poster: BOOLEAN
    -- Is there a playboy/girl poster on the door?
  open
    -- Opens the door
  do
    -- Implementation of open
  end
  -- more features...
end
```

# Finding the right abstractions (classes)

---



How would you model...

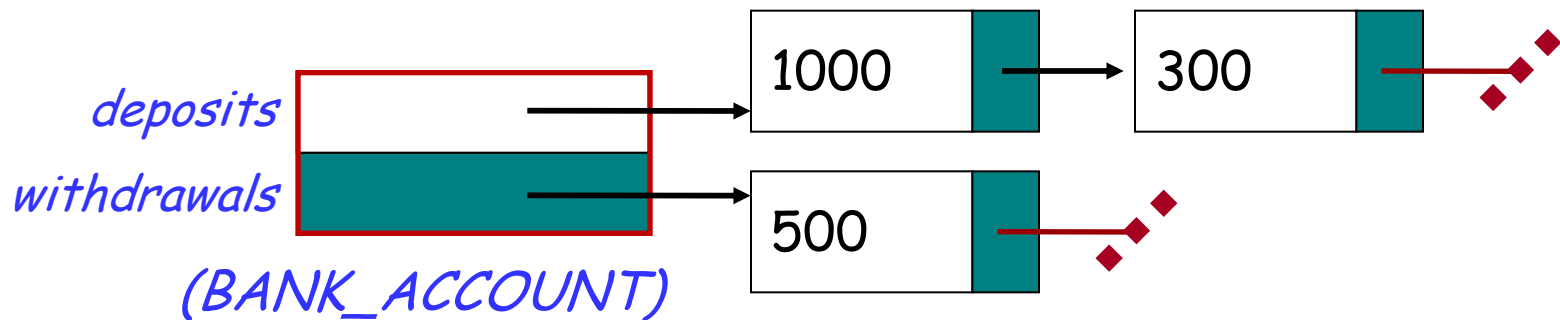
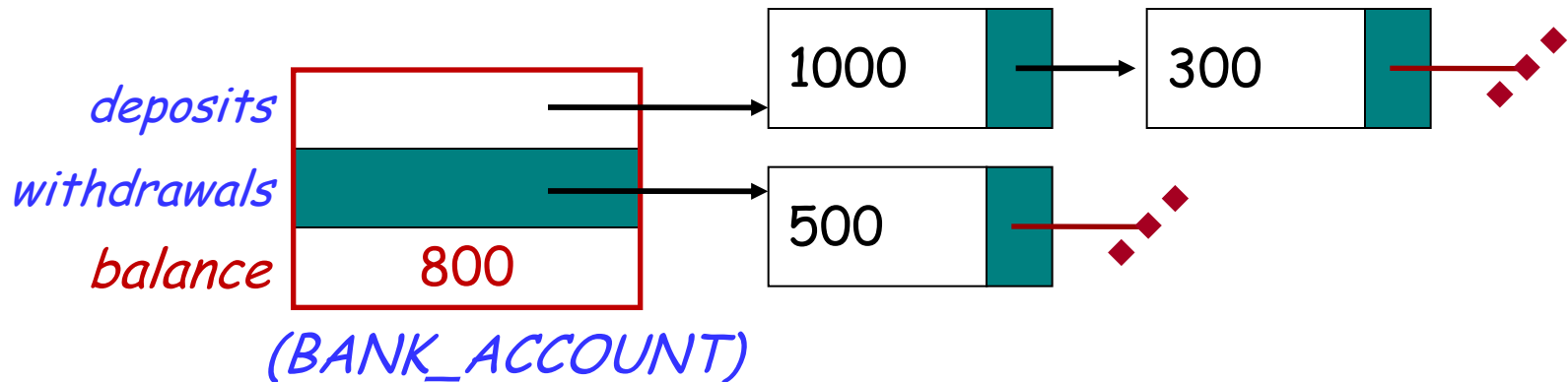
... the computer?

... the bed?

How would you model an elevator in a building?

**Hands-On**

# Finding the right abstractions (features)



**invariant:**  $balance = total\ (deposits) - total\ (withdrawals)$

Which one would you choose and why?

# Uniform access principle

---



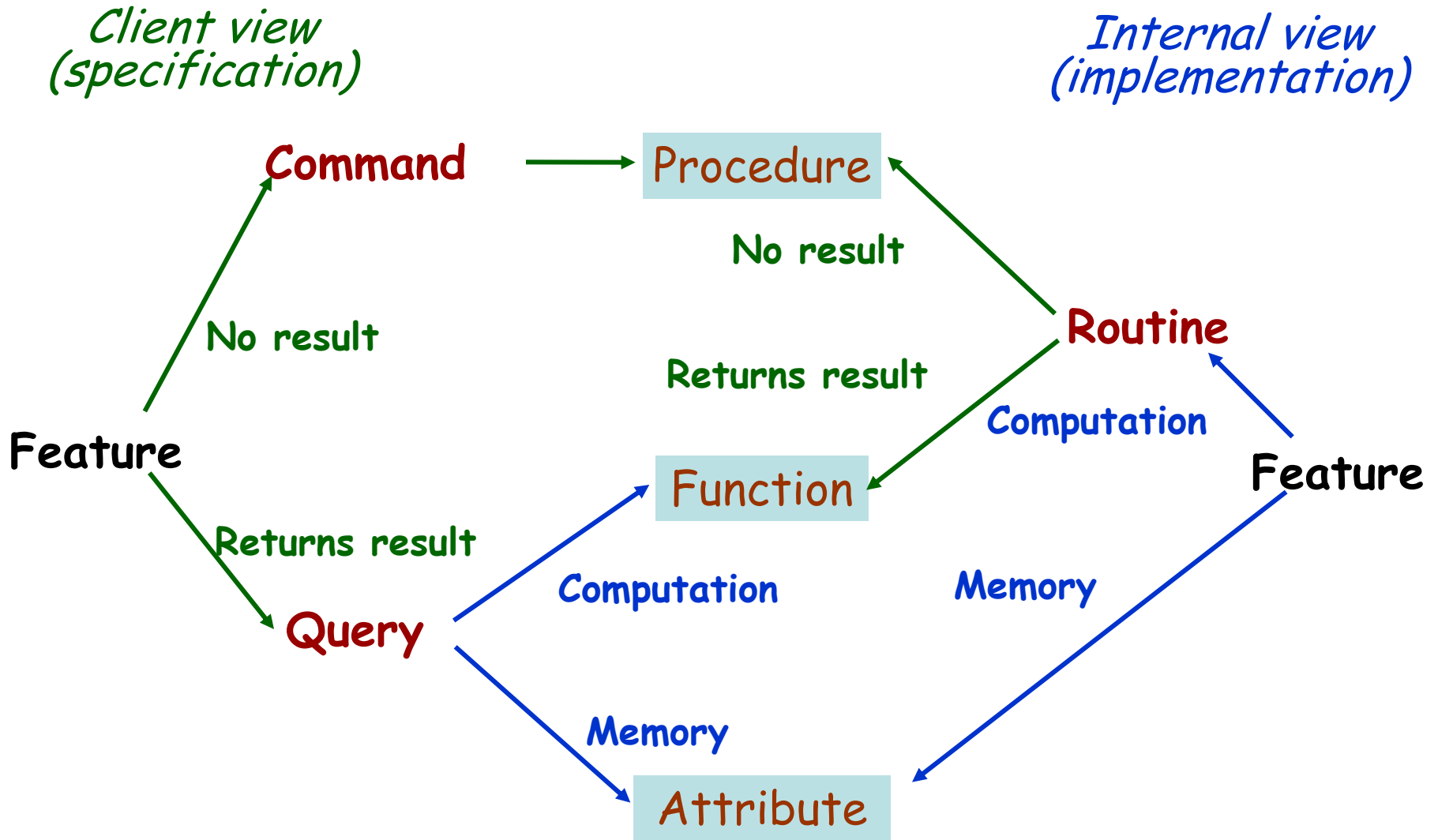
The client is interested in **what** a service does, not how it does it.

It doesn't matter for the client, whether you **store** or **compute**, he just wants to obtain the *balance*.

Features should be accessible to clients the same way, no matter whether they are implemented by storage or computation

*my\_account.balance*

# Features: the full story (again...)



# Two kinds of routines

---



## Procedure

- from the client's viewpoint it is a **command**
- call is an **instruction**

## Function

- from the client's viewpoint is a **query**
- call is an **expression**

# Naming conventions

---



Names for **classes**:

*PASSENGER, STUDENT, NUMERIC, STORABLE*

Names for **queries**:

*balance, name, first\_element, list\_of\_students*

➤ for **boolean queries**:

*full, after, is\_empty, is\_best\_choice*

Names for **commands**:

*run, do\_nothing, pimp\_my\_exersice\_session*

Composed of English words using underscore

Full English words, but short

# How do you like these names?

---



class *SOLVE\_QUADRATIC\_EQUATION*

feature

*solve (a, b, c: REAL) do ... end*

*get\_dscrm (a, b, c: REAL): REAL do ... end*

*how\_many\_solutions\_there\_are: INTEGER*

*first\_solution, second\_solution: REAL*

end

**Hands-On**

# I like these better!

---



**class** *QUADRATIC\_EQUATION\_SOLVER*

**feature**

*solve (a, b, c: REAL) do ... end*

*discriminant (a, b, c: REAL): REAL do ... end*

*solution\_count: INTEGER*

*first\_solution, second\_solution: REAL*

**end**

```
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.

# 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
end
```

OK: unqualified call

OK: exported to all

Error: not exported to PERSON

OK: unqualified call

# The export status does matter!

---



```
class PROFESSOR
```

```
  create
```

```
    make
```

```
  feature
```

```
    make (a_exam_draft: STRING)
```

```
      do
```

```
        exam_draft := a_exam_draft
```

```
      end
```

```
  feature
```

```
    exam_draft: STRING
```

```
end
```

# The export status does matter!

---



```
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
```

# 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** *:= prof.exam\_draft*

**end**

**end**

# The export status does matter!



*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*

AH HA HA HA HA!



# The export status does matter!

---



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
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
end
```



Invalid call!



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!

# Exporting attributes

---



**class** *TEMPERATURE*

**feature**

*celsius\_value, kelvin\_value: INTEGER*

*set\_celsius (a\_value: INTEGER)*

**require**

*above\_absolute\_zero: a\_value >= -273*

**do**

*celsius\_value := a\_value*

*kelvin\_value := celsius\_value + 273*

**end**

**invariant**

*above\_absolute\_zero: celsius\_value >= -273*

*correspond: kelvin\_value = celsius\_value + 273*

**end**

If you like the syntax

*x.f := 5*

you can declare an **assigner** for *f*

- In class *TEMPERATURE*  
*celsius\_value: INTEGER assign set\_celsius*

- In this case

*t.celsius\_value := 36*

is a shortcut for

*t.set\_celsius (36)*

- ... and it won't break the invariant!