



Einführung in die Programmierung Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 6



- Abstractions
- Exporting features
- Exercise: practicing contracts



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

A 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.



"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."

Finding the right abstractions (classes)



Suppose you want to model your room:

```
class ROOM
```

```
feature
```

```
-- to be determined
```

```
end
```

location door bed material 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 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...

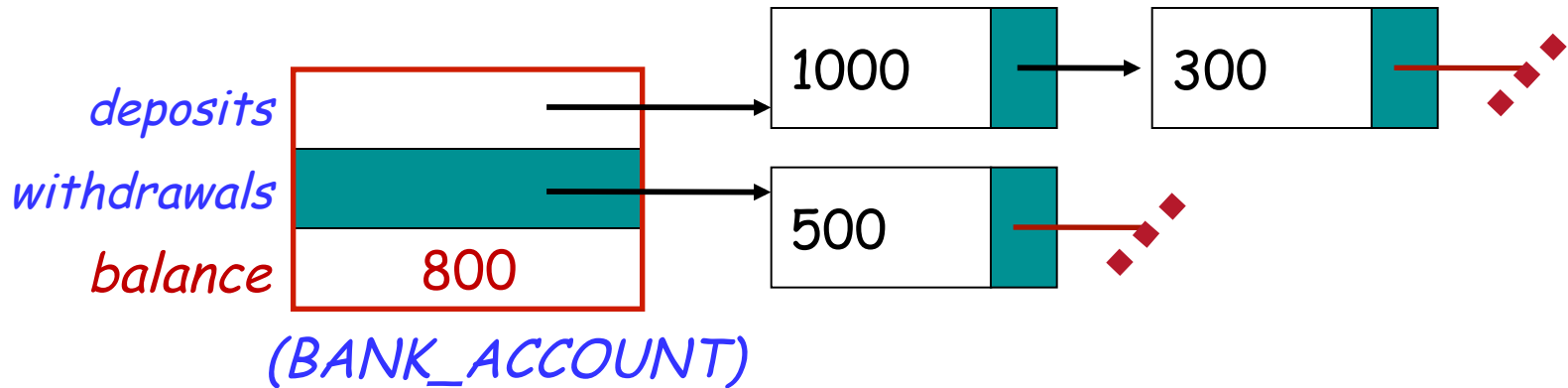
... the computer?

... the bed?

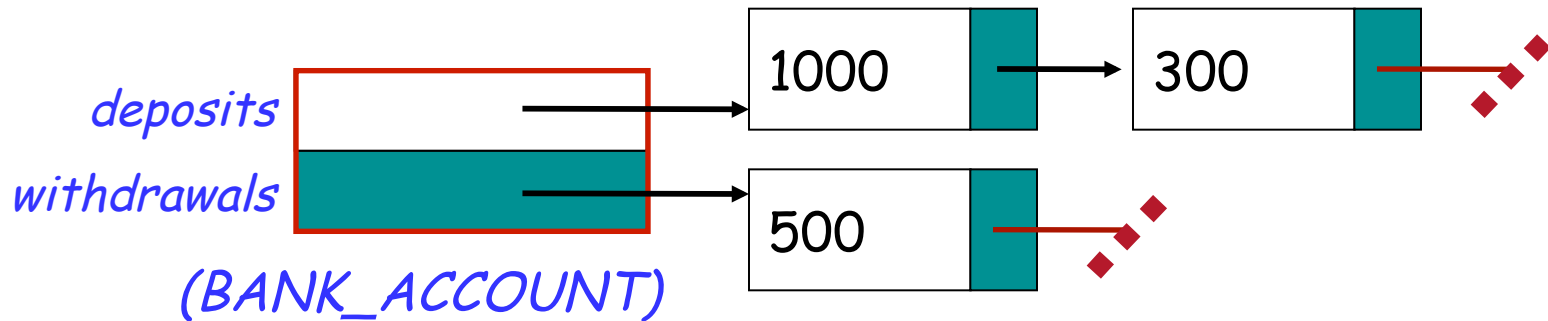
How would you model an elevator in a building?

Hands-On

Finding the right abstractions (features)



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



Which one would you choose and why?

Exporting features: The stolen exam



```
class ASSISTANT

  create
    make
  feature
    make (a_prof: PROFESSOR)
      do
        prof := a_prof
      end
  feature
    prof: PROFESSOR
  feature
    propose_draft (a_draft: STRING)
      do
        prof.review_draft(a_draft)
      end
  end
end
```

For your eyes only



```
class PROFESSOR
```

```
  create
```

```
    make
```

```
  feature
```

```
    make
```

```
      do
```

```
        exam_text := "exam is not ready"
```

```
      end
```

```
  feature
```

```
    exam_text: STRING
```

```
    review_draft (a_draft: STRING)
```

```
      do
```

```
        -- review 'a_draft' and put the result into 'exam_text'
```

```
      end
```

```
end
```

Exploiting a hole in information hiding



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

Don't try this at home!



you: STUDENT

your_prof: PROFESSOR

your_assi: ASSISTANT

stolen_exam: STRING

create your_prof.make

create your_assi.make (your_prof)

create you.make (your_prof, your_assi)

your_assi.propose_draft ("top secret exam!")

stolen_exam := you.stolen_exam

AH HA HA HA HA!



Secretive professor



```
class STUDENT
  create
    make
  feature
    make (a_assi: ASSISTANT)
      do
        assi := a_assi
      end
  feature
    assi: ASSISTANT
  feature
    stolen_exam: STRING
      do
        Result := assi.prof.exam_text
      end
  end
end
```

AH HA HA HA HA!





```
class      A
feature    g...
feature    {B, C}
           f...
end
```

Use selective export for the features

Fixing the issue



Hands-On

```
class PROFESSOR
create
  make
feature
  make
  do
    exam_text := "exam is not ready"
  end

feature {PROFESSOR, ASSISTANT}
  exam_text: STRING

  review_draft (a_draft: STRING)
  do
    -- review 'a_draft' and put the result into 'exam_text'
  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_text
  end
end
end
```



Invalid call!

```
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
end
```

OK: unqualified call

OK: exported to all

Error: not exported to PERSON

OK: unqualified 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!

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

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
    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
    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 ?

Specification of a card game



A deck is initially made of 36 cards

Every card in the deck represents a value in the range 2..10

Every card also represents 1 out of 4 possible colors

The colors represented in the game cards are:
red ('R'), white ('W'), green ('G') and blue ('B')

As long as there are cards in the deck, the players can look at the top card and remove it from the deck

Class CARD create make



Hands-On

make (a_color: *CHARACTER*, a_value: *INTEGER*)

-- Create a card given a color and a value.

require

...

ensure

...

color: *CHARACTER*

-- The card color.

value: *INTEGER*

-- The card value.

Class CARD: which colors are valid?



Hands-On

```
is_valid_color (a_color: CHARACTER): BOOLEAN
```

```
-- Is `a_color` a valid color?
```

```
require
```

```
...
```

```
ensure
```

```
...
```

Class CARD: which ranges are valid?



is_valid_range (n: *INTEGER*): *BOOLEAN*

-- Is `n` in the acceptable range?

require

...

ensure

...

invariant

...

Hands-On

Class CARD create make: reloaded



Hands-On

make (a_color: *CHARACTER*, a_value: *INTEGER*)

-- Create a card given a color and a value.

require

...

ensure

...

color: *CHARACTER*

-- The card color.

value: *INTEGER*

-- The card value.

Class DECK create make



Hands-On

make

-- Create a deck with random cards.

require

...

ensure

...

feature {*NONE*} -- Implementation

card_list: *LINKED_LIST*[*CARD*]

-- Deck as a linked list of cards.

Class DECK queries



Hands-On

top_card: *CARD*

-- The deck's top card.

is_empty: *BOOLEAN*

-- Is Current deck empty?

do

...

end

count: *INTEGER*

-- Number of remaining cards in the deck.

do

...

end

Removing the top card from DECK



`remove_top_card`

`-- Remove the top card from the deck.`

`require`

`...`

`ensure`

`...`

Hands-On

The class invariant



Hands-On

invariant

...