Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 7
Mock exam next week!

- Monday exercise groups: November 8
- Tuesday exercise groups: November 9
- You have to be present
- The week after we will discuss the results
- Assignment 7 due on November 16
Today

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

- 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.
“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.”

(extract from http://www.joelonsoftware.com/articles/LeakyAbstractions.html)
Discussing abstractions

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 \texttt{DOUBLE} is probably enough, since all we are interested in is it's value:

\begin{verbatim}
class ROOM
feature
  size: DOUBLE
  -- Size of the room.
end
\end{verbatim}
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 \texttt{BOOLEAN} will do:

\begin{verbatim}
class ROOM
feature
    size: DOUBLE
        -- Size of the room.
    is_door_open: BOOLEAN
        -- Is the door open or closed?
...
end
\end{verbatim}
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
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_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)

\[ \text{invariant: } \text{balance} = \text{total (deposits)} - \text{total (withdrawals)} \]

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...)

Client view (specification)

Command → Procedure
  - No result

Feature
  - Returns result

Query
  - No result

Internal view (implementation)

Procedure → Routine
  - Returns result

Routine
  - Computation

Function
  - Computation

Attribute
  - Memory

Feature
  - Memory

Computation
Two kinds of queries

Attribute

- from the client’s viewpoint it is a query
- call is an expression
- from the implementation’s viewpoint uses memory

Function

- from the client’s viewpoint is a query
- call is an expression
- from the implementation’s viewpoint uses computation
Exporting features

Status of calls in a client with \texttt{a1} of type \texttt{A}:

- \texttt{a1.f, a1.g}: valid in any client
- \texttt{a1.h}: invalid everywhere (including in \texttt{A}'s text!)
- \texttt{a1.j}: valid in \texttt{B, C} and their descendants (invalid in \texttt{A}!)
- \texttt{a1.m}: valid in \texttt{B, C} and their descendants, as well as in \texttt{A} and its descendants.
class PERSON
feature
 name: STRING
feature {BANK}
 account: BANK_ACCOUNT
feature {NONE}
 loved_one: PERSON
 think
do
 end
lend_100_franks
do
 loved_one.account.transfer (account, 100)
end
The strange case of the stolen exam

class PROFESSOR

create

  make

feature

  make (an_exam_draft: STRING)
  do
    exam_draft := an_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; an_assi: ASSISTANT)
do

    prof := a_prof
    assi := an_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
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!
Information hiding vs. creation routines

```ruby
class PROFESSOR
create
  make
feature {None}
    make (an_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 (an_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? What if I have `create {NONE} make` instead of `create {COLLEGE_MANAGER} make`?
Exporting attributes

Exporting an attribute only means giving read access

Attributes of other objects can be changed only through commands

- protecting the invariant
- no need for getter functions!
class \textit{TEMPERATURE} \\
\textbf{feature} \\
\textit{celsius\_value: INTEGER} \\
\textbf{make\_celsius} (\textit{a\_value: INTEGER}) \\
\textbf{require} \\
\textit{above\_absolute\_zero: a\_value >= - Celsius\_zero} \\
\textbf{do} \\
\textit{celsius\_value := a\_value} \\
\textbf{ensure} \\
\textit{celsius\_value\_set := celsius\_value = a\_value} \\
\textbf{end} \\
\textbf{end}
Assigners

If you like the syntax

\[ x.f := 5 \]

you can declare an assigner for \( f \)

- In class \( \text{TEMPERATURE} \)

  \( \text{celsius\_value: INTEGER assign make\_celsius} \)

- In this case

  \[ t.\text{celsius\_value} := 36 \]

  is a shortcut for

  \[ t.\text{make\_celsius} (36) \]

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