Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 4
Problems in Assignment-2 Solutions

- **Command or query?**
  - `connecting_lines`
    
    
    (a_station_1, a_station_2: STATION): V_SEQUENCE [LINE]
  - Noun phrases for query names; verb phrases for command names

- **Instruction separation?**
  - Comma (,), space( ), semicolon (;), or nothing

- **STRING_8 Vs. STRING_32**

```
make
local
  l_line: STRING_32
  c: UTF_CONVERTER

do
  Io.read_line
  l_line := c.utf_8_string_8_to_string_32 (Io.last_string)
  print (l_line.count)
end
```
Today

- Understanding contracts (preconditions, postconditions, and class invariants)
- Reference types vs. expanded types
- Basic types
- Entities and objects
- Object creation
- Assignment
Why do we need contracts at all?

- They are executable specifications that evolve together with the code
  - Together with tests, they are a great tool for finding bugs
  - They help us to reason about an O-O program at the level of classes and routines
  - Proving (part of) programs correct requires some way to specify the way the program should operate. Contracts are a way to specify the program
Assertions

When the condition is violated, the assertion tag (if present) is used to construct a more informative error message.

Assertion tag (optional, but recommended)

balance_non_negative: balance >= 0

Assertion clause

Condition (required)

When the condition is violated, the assertion tag (if present) is used to construct a more informative error message.
Property that a feature imposes on every client

\[ \text{clap} \left( n: \text{INTEGER} \right) \]

\[ -- \text{Clap } n \text{ times and update } count. \]

\[ \text{require} \]

\[ \text{not\_too\_tired: count} \leq 10 \]
\[ \text{n\_positive: } n > 0 \]

A feature with no \textbf{require} clause is always applicable, as if the precondition reads

\text{require}

\text{always\_OK: True}
Property that a feature guarantees on termination

\[ \text{clap} \left( n: \text{INTEGER} \right) \]

\[ \text{-- Clap } n \text{ times and update } count. \]

\[ \text{require} \]

\[ \text{not\_too\_tired: } count \leq 10 \]
\[ \text{n\_positive: } n > 0 \]

\[ \text{ensure} \]

\[ \text{count\_updated: } count = \text{old count} + n \]

A feature with no \textbf{ensure} clause always satisfies its postcondition, as if the postcondition reads

\[ \text{ensure} \]

\[ \text{always\_OK: True} \]
### Class Invariant

Property that is true of the current object at any *observable* point

```plaintext
class ACROBAT
...

invariant
  count_non_negative: count >= 0
end
```

A class with no *invariant* clause has a trivial invariant

```
always_OK: True
```
Add pre- and postconditions to:

\[ \text{smallest\_power} \ (n, \text{bound}: \text{NATURAL}): \text{NATURAL} \]
\[ \text{-- Smallest x such that } n' \text{ \^} x \text{ is greater or equal } \text{\`bound'} . \]

require

???
do

...

ensure

???
end
One possible solution

\[ \text{smallest\_power}(n, \text{bound}: \text{NATURAL}): \text{NATURAL} \]

\[ -- \text{Smallest } x \text{ such that } n'^x \text{ is greater or equal } \text{`bound'}. \]

\[ \text{require} \]
\[ n\_\text{large\_enough}: n > 1 \]
\[ \text{bound\_large\_enough}: \text{bound} > 1 \]
\[ \text{do} \]
\[ \ldots \]
\[ \text{ensure} \]
\[ \text{greater\_equal\_bound}: n ^ \text{Result} \geq \text{bound} \]
\[ \text{smallest}: n ^ (\text{Result} - 1) < \text{bound} \]
\[ \text{end} \]
Hands-on exercise

Add invariant(s) to the class\textit{ACROBAT\_WITH\_BUDDY}.

Add preconditions and postconditions to feature \textit{make} in \textit{ACROBAT\_WITH\_BUDDY}.
Class ACROBAT_WITH_BUDDY

class ACROBAT_WITH_BUDDY

inherit ACROBAT

redefine twirl, clap, count

create make

feature

make (p: ACROBAT)
do
  -- Remember `p' being the buddy.
end

clap (n: INTEGER)
do
  -- Clap `n' times and forward to buddy.
end

twirl (n: INTEGER)
do
  -- Twirl `n' times and forward to buddy.
end

count: INTEGER
do
  -- Ask buddy and return his answer.
end

buddy: ACROBAT
end
What are reference and expanded types?

**Reference types:** $s$ contains the address (reference, or location), of the object.

Example:

$$s: \text{STATION}$$

**Expanded types:** $p$ points directly to the object.

Example:

$$p: \text{POINT}$$
Why expanded types?

- Representing basic types (*INTEGER*, *REAL*, ...)

- Modeling external world objects realistically, i.e. describing objects that have sub-objects (and no sharing), for example a class *WORKSTATION* and its *CPU*. 
How to declare an expanded type

To create an expanded type, declare the class with keyword `expanded`:

expanded class `COUPLE`

feature -- Access

  `man, woman : HUMAN`
  `years_together : INTEGER`

end

Now all the entities of type `COUPLE` will automatically become expanded:

  `pitt_and_jolie : COUPLE`
Objects of reference or expanded types

Objects of reference types: they don't exist when we declare them (they are initially Void).

\( s: \text{STATION} \)

We need to explicitly create them with a create instruction.

\texttt{create s}

Objects of expanded types: they exist by just declaring them (they are never Void)

\( p: \text{POINT} \)

Feature \texttt{default_create} from ANY is implicitly invoked on them
Can expanded types contain reference types?

Expanded types can contain reference types, and vice versa.

\[
pitt\_and\_jolie\rightarrow (\text{SOME\_CLASS}) \rightarrow (\text{HUMAN})\]
Reference equality

1.0 2.0
(VECTOR)

1.0 2.0
(VECTOR)

1.0 2.0
(VECTOR)

1.0 2.0
(VECTOR)

a = b?
True

a = b?
False
Expanded entities equality

Entities of expanded types are compared by value!

\[ a = b ? \]

True
Expanded entities equality

\[(SOME\_CLASS)\]

\[a = b?\]

Hands-On

\[a\]

\[b\]

\[
\begin{array}{c}
\text{John} \\
32 \\
\text{(HUMAN)}
\end{array}
\]

\[
\begin{array}{c}
\text{Jane} \\
30 \\
\text{(HUMAN)}
\end{array}
\]

\[
\begin{array}{c}
10 \\
\text{(COUPLE)}
\end{array}
\]

\[
\begin{array}{c}
\text{John} \\
32 \\
\text{(HUMAN)}
\end{array}
\]

\[
\begin{array}{c}
\text{Jane} \\
30 \\
\text{(HUMAN)}
\end{array}
\]

\[
\begin{array}{c}
10 \\
\text{(COUPLE)}
\end{array}
\]
Expanded entities equality

\[ a = b? \]

\( \text{(SOME\_CLASS)} \)

\( \text{(COUPLE)} \)

\( \text{(HUMAN)} \)

\( \text{John} \)

\( 32 \)

\( \text{Jane} \)

\( 30 \)
Basic types

Their only privilege is to use manifest constants to construct their instances:

\[ b: \text{BOOLEAN} \]
\[ x: \text{INTEGER} \]
\[ c: \text{CHARACTER} \]
\[ s: \text{STRING} \]

... 
\[ b := \text{True} \]
\[ x := 5 \quad \text{-- instead of create } x.\text{make_five} \]
\[ c := 'c' \]
\[ s := "I love Eiffel" \]
Some basic types (BOOLEAN, INTEGER, NATURAL, REAL, CHARACTER) are expanded...

\[ a := b \]

... and immutable (they do not contain commands to change the state of their instances)...

\[ a := a.\text{plus} \, (b) \]

Instead of \[ a.\text{add} \, (b) \]

Alias for \text{plus}
Strings are a bit different

Strings in Eiffel are *not* expanded...

\[ s: \text{STRING} \]

... and *not* immutable

\[ s := \text{“I love Eiffel"} \]
\[ s.\text{append}(\text{“ very much!”}) \]
Object comparison: = versus ~

s1: STRING = “Teddy”
}s2: STRING = “Teddy”
...

s1 = s2 -- False: reference comparison on different objects

s1 ~ s2 -- True
...

Now you know what to do if interested in comparing the content of two objects
Initialization

Default value of any reference type is Void

Default values of basic expanded types are:

- **False** for BOOLEAN
- 0 for numeric types (INTEGER, NATURAL, REAL)
- “null” character (its code is 0) for CHARACTER

Default value of a non-basic expanded type is an object, whose fields have default values of their types
Initialization

What is the default value for the following classes?

expanded class \textit{POINT}
feature \textit{x}, \textit{y}: REAL end

class \textit{VECTOR}
feature \textit{x}, \textit{y}: REAL end

\textit{STRING}

\begin{tabular}{|c|c|}
\hline
\textit{x} & \textbf{0.0} \\
\hline
\textit{y} & \textbf{0.0} \\
\hline
\end{tabular}

(POINT)

Void

Void
Creation procedures

- Instruction `create x` will initialize all the fields of the new object attached to `x` with default values
- What if we want some specific initialization? E.g., to make object consistent with its class invariant?

Class `CUSTOMER`
...

```
    id : STRING
```

invariant
```
    id /= Void
```

- Use creation procedure:

  ```
  create a_customer.set_id("13400002")
  ```
class CUSTOMER

create set_id

feature

id : STRING
    -- Unique identifier for Current.

set_id (a_id : STRING)
    -- Associate this customer with `a_id'.
    require
      id_exists : a_id /= Void
    do
      id := a_id
    ensure
      id_set : id = a_id
    end

invariant

id_exists : id /= Void
end
Object creation

To create an object:

- If class has no `create` clause, use basic form:
  \[
  \text{create } x
  \]

- If the class has a `create` clause listing one or more procedures, use
  \[
  \text{create } x\.\text{make}(\ldots)
  \]
  where `make` is one of the creation procedures, and `\ldots` stands for arguments if any.
class DIRECTOR
create prepare_and_play
feature
  acrobat1, acrobat2, acrobat3: ACROBAT
  friend1, friend2: ACROBAT_WITH_BUDDY
  author1: AUTHOR
  curmudgeon1: CURMUDGEON

prepare_and_play
do
  author1.clap (4)
  friend1.twirl (2)
  curmudgeon1.clap (7)
  acrobat2.clap (curmudgeon1.count)
  acrobat3.twirl (friend2.count)
  friend1.buddy.clap (friend1.count)
  friend2.clap (2)
end
end
Some acrobatics

class DIRECTOR
create prepare_and_play
feature
  acrobat1, acrobat2, acrobat3: ACROBAT
  friend1, friend2: ACROBAT_WITH_BUDDY
  author1: AUTHOR
  curmudgeon1: CURMUDGEON

prepare_and_play
  do
  1  create acrobat1
  2  create acrobat2
  3  create acrobat3
  4  create friend1.make_with_buddy(acrobat1)
  5  create friend2.make_with_buddy(friend1)
  6  create author1
  7  create curmudgeon1
  end
end

Which entities are still Void after execution of line 4?
Which of the classes mentioned here have creation procedures?
Why is the creation procedure necessary?
Custom initialization for expanded types

- Expanded classes are not creatable using a creation feature of your choice
  ```
  expanded class POINT
  create make
  feature make do x := 5.0; y := 5.0 end ...
  end
  ```

- But you can use (and possibly redefine) default_create
  ```
  expanded class POINT
  inherit ANY
  redefine default_create
  feature
    default_create do
      x := 5.0; y := 5.0
    end
  end
  ```
Assignment is an instruction (What other instructions do you know?)

Syntax:

\[ a := b \]

- where \( a \) is a variable (e.g., attribute) and \( b \) is an expression (e.g. argument, query call);
- \( a \) is called the target of the assignment and \( b \) the source.

Semantics:

- after the assignment \( a \) equals \( b \) (\( a = b \));
- the value of \( b \) is not changed by the assignment.
Reference assignment

\[ a := b \]

\( a \) references the same object as \( b \):

\[ a = b \]
Expanded assignment

\begin{align*}
a &:= b \\
\text{The value of } b \text{ is copied to } a, \text{ but again: } \\
& a = b
\end{align*}
Explain graphically the effect of an assignment:

Here \textit{COPPLE} is an expanded class, \textit{HUMAN} is a reference class.
Attachment

- More general term than assignment
- Includes:
  - Assignment
    
    \[ a := b \]

  - Passing arguments to a routine
    \[
    f(a: SOME\_TYPE) \\
    \text{do ... end}
    \]

    \[ f(b) \]

  - Same semantics
Dynamic aliasing

\( a, b: \) VECTOR

... create \( b\).make (1.0, 0.0)
\( a := b \)

- now \( a \) and \( b \) reference the same object (they are two names or aliases of the same object)
- any change to the object attached to \( a \) will be reflected when accessing it using \( b \)
- any change to the object attached to \( b \) will be reflected when accessing it using \( a \)
Dynamic aliasing

What are the values of $a.x$, $a.y$, $b.x$ and $b.y$ after executing instructions 1-4?

$a, b$: VECTOR

... create $a.make$ (-1.0, 2.0)

1 create $b.make$ (1.0, 0.0)

2 $a := b$

3 $b.set_x$ (5.0)

4 $a.set_y$ (-10.0)
Meet Teddy