Assignment 7: Inheritance

ETH Zurich

Hand-out: 11 December 2006
Due: 19 December 2006

Calvin and Hobbes© Bill Watterson

1 Dynamic exercise

Goal
Understand the effects of dynamic binding.

Summary
Have a look at the classes POLYGON and RECTANGLE shown in figure 1. Figure 2 shows how the vertices of a polygon are represented.

Given the declarations

\[ p: \text{POLYGON}; r: \text{RECTANGLE}; x: \text{REAL} \]

you can perform the following feature calls and assignments:

\[ x := p.\text{perimeter} \]
\[ x := r.\text{perimeter} \]
\[ x := r.\text{diagonal} \]
\[ p := r \quad \text{(see figure 3)} \]
These ones are NOT permitted:

\[ x := p \text{.diagonal} \quad (\text{even if it occurs immediately after } p := r) \]
\[ r := p \]

```plaintext
class POLYGON
create
  make
feature
  vertices : ARRAY [POINT]
  vertices_count : INTEGER
  perimeter : REAL is
    -- Perimeter length
    do
      from ... until ... loop
        Result := Result + (vertices [i]).distance (vertices [i + 1])
      end
    ... end
  invariant
    vertices_count \geq 3
    vertices_count = vertices.count
end

class RECTANGLE
inherit POLYGON
redefine
  perimeter
end
create
  make
feature
  diagonal, side1, side2 : REAL
  perimeter : REAL is
    -- Perimeter length
    do
      Result := 2 * (side1 + side2)
    end
  invariant
    vertices_count = 4
end
```

Figure 1: Classes POLYGON and RECTANGLE
Examples

Consider the inheritance hierarchy in figure 4 and the corresponding classes in figures 5, 6, and 7.
Question 1: Is the following code valid? Explain why.

\[ \begin{align*}
    b & : \text{BOOK} \\
    \text{create} & \ b.\text{make} \\
    b.\text{print\_book} \\
\end{align*} \]

Answer 1: Yes, because \( b \) is of type \( \text{BOOK} \) and class \( \text{BOOK} \) has a feature \( \text{print\_book} \).

Question 2: The code presented in question 1 is valid. What message is printed when executing this code?

Answer 2: “This is a book.”

```plaintext
class BOOK
create
make

feature -- Initialization
    make is
        -- Initialize book.
        do
        end

feature -- Output
    print\_book is
        -- Print message.
        do
            io.\text{put\_string} (" This is a book.\%N")
        end

end
```

Figure 5: Class BOOK
class  
  TEXTBOOK

inherit
  BOOK
  rename
    print_book as print_textbook
  redefine
    print_textbook
end

create
  make

figure -- Output

  print_textbook is
    -- Print message.
    do
      io.put_string ('"This is a textbook.%N"
    end

end

Figure 6: Class TEXTBOOK

class
  COMICS

inherit
  BOOK
  rename
    print_book as print_comics
  redefine
    print_comics
end

create
  make

feature -- Output

  print_comics is
    -- Print message.
    do
      Precursor{BOOK}
      io.put_string ('"This is a comics.%N"
    end

end

Figure 7: Class COMICS
To do

1. Is the following code valid? Explain why.
   
   ```
   b: BOOK
   create {TEXTBOOK} b.make
   b.print_book
   ```

2. Is the following code valid? Explain why.
   
   ```
   b: BOOK
   create {TEXTBOOK} b.make
   b.print_textbook
   ```

3. One of the code samples presented in question 1 or 2 is valid. What message is printed when executing this code?

4. Is the following code valid? Explain why.
   
   ```
   b: BOOK
   t: TEXTBOOK
   create t.make
   b := t
   b.print_book
   ```

5. Is the following code valid? Explain why.
   
   ```
   b: BOOK
   t: TEXTBOOK
   create t.make
   b := t
   b.print_textbook
   ```

6. One of the code samples presented in question 4 or 5 is valid. What message is printed when executing this code?

7. Is the following code valid? Explain why.
   
   ```
   b: BOOK
   c: COMICS
   create {COMICS} b.make
   c ?= b
   if c /= Void then c.print_book end
   ```

8. Is the following code valid? Explain why.
   
   ```
   b: BOOK
   c: COMICS
   create {COMICS} b.make
   c ?= b
   if c /= Void then c.print_comics end
   ```

9. One of the code samples presented in question 7 or 8 is valid. What message is printed when executing this code?
To hand in

Hand in your answers to questions 1 to 9.

2 Inherited Fraction

Goal

• Inherit from a class.
• Use the infix and the prefix notation for features.

Description

NUMERIC is a deferred class providing the following features:

• one
• zero
• divisible
• exponentiable
• infix "\+"
• infix "\-
• infix "/"
• infix "\*
• prefix "\+"
• prefix "\-

Your task is to implement class FRACTION which represents fractions of the form \[ \frac{\text{numerator}}{\text{denominator}} \]. FRACTION inherits from NUMERIC. The class FRACTION_TEST shown in figure should work with your class without any changes. You can download the source of FRACTION_TEST from

http://se.inf.ethz.ch/teaching/ws2006/0001/exercises/fraction_test.e

To do

1. Option 1: Solving the exercise with Tootor.
   (a) Download and install the tool. You will find the link on the course website. The instructions on how to install and use Tootor are provided there.
   (b) Launch Tootor and choose the Fraction exercise.
   (c) With Tootor’s guidance, work on the tasks it presents until the whole exercise is solved.
2. Option 2: Writing a solution from scratch in EiffelStudio.
   (a) Create a new project with the root class `FRACTION_TEST`.
   (b) Copy and paste the class text in the root class.
   (c) Add a new class `FRACTION`, inherit from `NUMERIC`, and implement the missing features.

```eiffel
class FRACTION_TEST
create
make
feature -- Initialization
   a, b, c: FRACTION
   make is
      -- Test the class FRACTION.
      do
         create a.make (1, 2)
         create b.make (3, 4)
         io.put_string ("Calculating with fractions:" + "%N%N")
         io.put_string ("a : " + a.out)
         io.put_string ("b : " + b.out)
         c := a + b
         io.put_string ("a + b : " + c.out)
         c := a - b
         io.put_string ("a - b : " + c.out)
         c := a * b
         io.put_string ("a * b : " + c.out)
         c := a / b
         io.put_string ("a / b : " + c.out)
      end
end
```

Figure 8: Class FRACTION_TEST

Hint
- To reduce a fraction, you can use a Greatest Common Divisor (GCD) algorithm, for example the Euclidian algorithm shown in the lecture.
- In Eiffel, integer division is done with `//`, integer remainder (modulo) with `\`.
- Have a closer look at class `FRACTION_TEST` for guidance on how to implement `FRACTION`.
Remarks

Do not forget contracts. $FRACTION$ has an obvious invariant.

To hand in

Hand in the source code of your class $FRACTION$. Make sure to upload your logs (Tootor logs if you used the tool; EiffelStudio logs if you did not use it). Thanks a lot!

3 Landing on your... feet

Goal

Understand what happens to contracts in the context of inheritance.

Description

When a routine is redefined in a subclass, its precondition may be kept or weakened; its postcondition may be kept or strengthened. Hence, in a redefined routine, any precondition is introduced with $\text{require else}$; any postcondition with $\text{ensure then}$. The resulting assertions have the following meaning:

- precondition: $\text{original\_precondition or new\_precondition}$
- postcondition: $\text{original\_postcondition and new\_postcondition}$

Class invariants are accumulated, i.e. every class inherits all the invariant clauses of its parents; these clauses are conceptually “and”-ed.

Assume you have a class $\text{PLANE}$ whose code is partly shown in figure 9. Planes have sensors which detect the altitude at which they are and if they are above earth or water. As the contract of routine $\text{land}$ shows, planes can land only under certain circumstances.

Hydroplanes are a special kind of plane. They can land on and take off from both earth and water. To land on water, they must deploy the flotation. Class $\text{HYDROPLANE}$ is as a subclass of $\text{PLANE}$; it redefines the feature $\text{land}$ to reflect the different landing ability of hydroplanes.

To do

Write the class $\text{HYDROPLANE}$ as a subclass of $\text{PLANE}$. Redefine the contracts of $\text{land}$ (leaving its body blank) and add any features needed in the contracts. Try to come up with the most precise contracts.

To hand in

Hand in the source code of $\text{HYDROPLANE}$. 
class PLANE

feature -- Status

  below: STRING
  -- What is below the plane

  altitude: INTEGER
  -- Altitude of the plane

... feature -- Basic operations

  land is
  -- Land
  require
    earth below: below is_equal ("earth")
  do
    ...
  ensure
    zero altitude: altitude = 0

end

Figure 9: Skeleton of class PLANE