Exercise 5: Multiple and repeated inheritance

Hand-out: 14 May 2004
Due: 28 May 2004

Master Solution

Summary: Inheritance, Agents

Name clashes under multiple inheritance
Name clashes may occur in case of multiple inheritance. To solve them, Eiffel has five feature adaptation clauses:

- **rename**: to change the name of a parent feature.
- **export**: to modify the visibility of a parent feature.
- **undefine**: to make a parent feature deferred.
- **redefine**: to redefine the signature and/or the body of a parent feature.
- **select**: to select a parent feature.

Deferred classes and features
A feature is either deferred or effective. To effect an inherited feature (deferred in the parent) is to make it effective. No need for a redefine clause.
Like a feature, a class is either deferred or effective.
A class is deferred if it has at least one deferred feature (possibly coming from an ancestor) that it does not effect. It is effective otherwise.
A deferred class may not be instantiated.
BUT: A deferred class may have assertions (in particular, a deferred routine may have a precondition and a postcondition, and the class may have a class invariant).

Agents
An agent expression is of the form:

```
agent your_function (?, u, v)
```

where ? corresponds to an open argument (i.e. set at the time of any call to the agent) and u and v correspond to close arguments (i.e. set at the time of the agent’s definition).

It is also possible to specify a target, as follows:

```
agent some_object.some_routine (?, u, v)
```

where some_object is the target.
In the above example, the target is closed (i.e. it is *some_object.*) The target may also be open, like in:

```plaintext
my_employee_list.for_all (agent {EMPLOYEE}.is_married)
```

“{EMPLOYEE}” is the open target.

### 1. Multiple and repeated inheritance

Consider the following class diagram:

Here are the texts of classes `DRIVER`, `SWISS_DRIVER`, and `FRENCH_DRIVER`:

```plaintext
class DRIVER

feature -- Access

  driver_license: DRIVER_LICENSE
  -- License of current driver

  violation_count: INTEGER
  -- Number of times the driver has violated
  -- the highway code

feature -- Basic operations

  pay_fee is
  -- Pay fee depending on the infringement done.

  require

  at_least_one_violation: violation_count >= 1

  do
  -- Do something here.

end

class SWISS_DRIVER

class FRENCH_DRIVER

class SWISS_FRENCH_DRIVER
```

Here are the texts of classes `DRIVER`, `SWISS_DRIVER`, and `FRENCH_DRIVER`:
**pay_tax** is
  -- Pay road tax.
  do
    -- Do something here.
  end

**invariant**

violation_count_is_positive: violation_count >= 0
end

**Class**

**SWISS_DRIVER**

**inherit**

**DRIVER**

**redefine**

  **payFee**,  
  **pay_tax**,  
end

**feature** -- Basic operations

pay_fee is
  -- Pay fee depending on the infringement done.
  do
    Precursor {**DRIVER**}
    **pay_swiss_fee**
  end

pay_swiss_fee is
  -- Pay swiss fee depending on the infringement done.
  do
    -- Do something more here.
    End

pay_tax is
  -- Pay road tax.
  do
    Precursor {**DRIVER**}
    **pay_swiss_tax**
  end

pay_swiss_tax is
  -- Pay swiss tax depending on `tax_payed`.
  do
    -- Do something more here.
  end

**feature** -- Basic queries

**tax_payed**: BOOLEAN
  -- Has the swiss driver already payed the annual tax?
end
To do

- Why, in Eiffel, any case of multiple inheritance is also a case of repeated inheritance?
- Write a possible implementation of class SWISS_FRENCH_DRIVER.

Hint

There are at least two solutions for class SWISS_FRENCH_DRIVER.
To hand in
Hand in the answer to the question on the previous page and the text of class `SWISS_FRENCH_DRIVER` and send the Eiffel file “swiss_french_driver.e” and the “.ace” file (in your project directory) to your assistant. Make sure that your project compiles. If it does not compile, explain your problems to your assistant.

Solution
- Why, in Eiffel, any case of multiple inheritance is also a case of repeated inheritance?

Any Eiffel class inherits from the class `ANY`.

Let’s take an example: if a class `C` inherits from two classes `A` and `B` (case of multiple inheritance):

In fact we have the following picture because `A` and `B` inherit from `ANY` (implicitly):

which means it is a case of repeated inheritance.

- Write a possible implementation of class `SWISS_FRENCH_DRIVER`.
First solution:

class

    SWISS_FRENCH_DRIVER

inherit

    SWISS_DRIVER
        redefine
            pay_fee,
            pay_tax
        end

    FRENCH_DRIVER
        undefine
            pay_fee,
            pay_tax
        end

feature -- Basic operations

    pay_fee is
        -- Pay fee depending on the infringement done.
        do
            Precursor {SWISS_DRIVER}
            pay_french_fee
        end

    pay_tax is
        -- Pay road tax.
        do
            Precursor {SWISS_DRIVER}
            pay_french_tax
        end

end

Second solution:

class

    SWISS_FRENCH_DRIVER
inherit

SWISS_DRIVER
    export {NONE}
    pay_swiss_fee,
    pay_swiss_tax
    redefine
    pay_fee,
    pay_tax
end

FRENCH_DRIVER
    export {NONE}
    pay_french_fee,
    pay_french_tax
    redefine
    pay_fee,
    pay_tax
end

DRIVER
    redefine
    pay_fee,
    pay_tax
end

feature -- Basic operation

pay_fee is
    -- Pay fee depending on the infringement done.
    do
        Precursor {DRIVER}
        pay_swiss_fee
        pay_french_fee
    end

pay_tax is
    -- Pay road tax.
    do
        Precursor {DRIVER}
        pay_swiss_fee
        pay_french_fee
    end
end
2. Deferred classes, Assertion inheritance

Consider the following classes A and B:

```plaintext
defered class
  A
feature – Access
    index: INTEGER
      -- Index
feature -- Element change
  change_index is
    -- Change `index’.
    require
      valid_index: index > 2
defered
    ensure
      index_changed: index > 10
end
class
  B
inherit
  A
    redefine
      change_index
end

feature -- Element change
  change_index is
    -- Change `index’.
    require
      valid_index: index > 5
    do
      -- Do something.
```

---

**ETHZ D-INFK**

**Programming in the large – Exercises**

**Prof. Dr. B. Meyer**

**Summer 2004**
To do

• What is the main difference between Java/C# interfaces and Eiffel deferred classes?
• Classes A and B would not compile. Explain why.
• Correct the text of classes A and B to make them compile.

Hint
Look at assertions.

To hand in
Hand in your answer to the above questions and the text of classes A and B.

Solution

• What is the main difference between Java/C# interfaces and Eiffel deferred classes?

Java/C# interfaces cannot contain any implementation at all: none of the features may be implemented; it cannot contain any attributes.
In Eiffel, a deferred class can contain attributes, and may be partially (or totally) implemented. A deferred class can also have assertions.

• Classes A and B would not compile. Explain why.

Class B is a descendant of A. Therefore the precondition of feature change_index should be introduced with the keyword require else (instead of just require) and the postcondition should be introduced with ensure then (instead of just ensure).

• Correct the text of classes A and B to make them compile. Explain why the precondition and postcondition of class B are “useless” in this case.

Replace require by require else and ensure by ensure then in feature change_index of the descendant class B.

Preconditions are “or-ed” and postconditions are “and-ed”. Therefore the actual contract of change_index in B is:

require
valid_index: index > 2 or else index > 5
ensure
index_changed: index > 10 and then index > 7
3. Agents

To do

- Write a class `EMPLOYEE` with at least the following information: name, age, salary, marital status, and gender. (Don’t forget the creation procedure and relevant setter procedures.)
- Write a class `CLIENT` (the root class), which creates a list of `EMPLOYEE`s with a few `EMPLOYEE`s in it and print the employees’ information using agents. You should display the following information:
  - Answer to the question: “Are all employees married?”
  - Answer to the question: “Are all employees women?”
  - Print the name, age, salary, marital status, and gender of all employees.

Hint

- You may need to add a few output features to the class `EMPLOYEE`.

To hand in

Hand in the text of classes `EMPLOYEE` and `CLIENT`.

Solution

- Possible class `EMPLOYEE`:

```plaintext
indexing

description: "Representation of an employee"

class

EMPLOYEE

create

make

feature {NONE} -- Initialization

make {a_name: like name; an_age: like age;
  a_salary: like salary; marital_status: like is_married;
  gender: like is_woman} is
  -- Set 'name' to 'a_name'.
  -- Set 'age' to 'an_age'.
  -- Set 'salary' to 'a_salary'.
  -- Set 'is_married' to 'marital_status'.
  -- Set 'is_woman' to 'gender'.

require

  a_name_not_void: a_name /= Void
  a_name_not_empty: not a_name.is_empty
  of_age: an_age > voting_age
  a_salary_positive: a_salary > 0

do

  name := a_name
  age := an_age
```
salary := a_salary
is_married := marital_status
is_woman := gender

ensure
name_set: name = a_name
age_set: age = an_age
salary_set: salary = a_salary
is_married_set: is_married = marital_status
is_woman_set: is_woman = gender

end

feature -- Access

name: STRING
-- Name of employee
age: INTEGER
-- Age of employee
salary: INTEGER
-- Salary of employee

Voting_age: INTEGER is 18
-- Voting age

feature -- Status report

is_married: BOOLEAN
-- Is employee married?

is_woman: BOOLEAN
-- Is employee a woman?

is_employee_married: BOOLEAN is
-- Is employee married? (This query is added because
-- of the current limitation on agents which are not
-- supported on attributes yet.)
do
Result := is_married
ensure
definition: Result = is_married
end

is_woman_employee: BOOLEAN is
-- Is employee a woman? (This query is added because
-- of the current limitation on agents which are not
-- supported on attributes yet.)
do
Result := is_woman
ensure
definition: Result = is_woman
end

feature -- Status setting

set_age (an_age: like age) is
-- Set 'age' to 'an_age'.
require
of_age: an_age > voting_age
do
  age := an_age
ensure
  age_set: age = an_age
end

set_salary (a_salary: like salary) is
  -- Set `salary' to `a_salary'.
require
  a_salary_positive: a_salary > 0
do
  salary := a_salary
ensure
  salary_set: salary = a_salary
end

set_is_married (a_value: like is_married) is
  -- Set `is_married' to `a_value'.
do
  is_married := a_value
ensure
  is_married_set: is_married = a_value
end

feature -- Output
  print_name is
    -- Print name of employee.
do
    io.put_string ("Name:%T")
    io.put_string (name)
    io.put_new_line
end

print_salary is
  -- Print salary of employee.
do
    io.put_string ("Salary:%T")
    io.put_integer (salary)
    io.put_new_line
end

print_age is
  -- Print age of employee.
do
    io.put_string ("Age:%T")
    io.put_integer (age)
    io.put_new_line
end

print_marital_status is
  -- Print marital status of employee.
do
    io.put_string ("Is employee married?%T")
    io.put_boolean (is_married)
    io.put_new_line
end
print_gender is
    -- Print marital status of employee.
    do
        io.put_string ("Is employee a woman? %T")
        io.put_boolean (is_woman)
        io.put_new_line
    end

invariant

    name_not_void: name /= Void
    name_not_empty: not name.is_empty
    of_age: age > voting_age
    salary_positive: salary > 0

end

• Possible class CLIENT:

indexing

    description: "Root class"

class

    CLIENT

create

    make

feature -- Initialization

    make is
        -- Create and initialize the list of `employees`
        -- and print information about `employees`.
        local
            employee1: EMPLOYEE
            employee2: EMPLOYEE
            employee3: EMPLOYEE
            employee4: EMPLOYEE
            employee5: EMPLOYEE
        do
            create employees.make
            -- Create 5 employees.
            create employee1.make ("Alice", 25, 4000, False, True)
            create employee2.make ("Bob", 36, 10000, False, False)
            create employee3.make ("John", 45, 7500, True, False)
            create employee4.make ("Judith", 22, 3000, False, True)
            create employee5.make ("Robert", 55, 9000, False, False)
            -- Add employees to the list of `employees`.
            employees.extend (employee1)
            employees.extend (employee2)
            employees.extend (employee3)
            employees.extend (employee4)
            employees.extend (employee5)
-- Print information about all 'employees'.
   print_employees
   end

feature -- Access
   employees: LINKED_LIST [EMPLOYEE] -- Employees

feature -- Output
   print_employees is
       -- Print information about 'employees'.
       do
           -- Are all employees married?
           io.put_string ("Are all employees married?%T")
           io.put_boolean (employees.for_all (
               agent {EMPLOYEE}.is_employee_married))
           io.put_new_line

           -- Are all employees women?
           io.put_string ("Are all employees women?%T")
           io.put_boolean (employees.for_all (
               agent {EMPLOYEE}.is_woman_employee))
           io.put_new_line
           io.put_new_line

           -- Print all information about 'employees':
           -- name, salary, age, marital status, gender.
           io.put_string ("Name of employees:%N")
           employees.do_all (agent {EMPLOYEE}.print_name)
           io.put_new_line

           io.put_string ("Salary of employees:%N")
           employees.do_all (agent {EMPLOYEE}.print_salary)
           io.put_new_line

           io.put_string ("Age of employees:%N")
           employees.do_all (agent {EMPLOYEE}.print_age)
           io.put_new_line

           io.put_string ("Marital status of employees:%N")
           employees.do_all (agent {EMPLOYEE}.print_marital_status)
           io.put_new_line

           io.put_string ("Gender of employees:%N")
           employees.do_all (agent {EMPLOYEE}.print_gender)
           io.put_new_line
       end

invariant
   employees_not_void: employees /= Void
   no_void_employee: not employees.has (Void)
end