

Classroom 1: Doubly linked fairy tales

ETH Zurich

Date: 6. December 2005

1 Terminology

Goal

This task will test your understanding of the O-O concepts presented so far in the lecture. This is a multiple-choice test.

Todo

Mark correct answers using the checkboxes. Multiple correct answers are possible; there is at least one correct answer per question. A correct answer is worth 1 point, an incorrect answer is worth -1 point. If the sum of your points is negative, you will receive 0 points.

1. A command...
 - a. is a query that is not implemented as an attribute.
 - b. may modify an object.
 - c. may appear in the precondition and the postcondition of another command but not in the precondition or the postcondition of a query.
 - d. may appear in the class invariant.
2. A query...
 - a. may be used as a creation procedure.
 - b. may be implemented as a routine.
 - c. may appear in the precondition and the postcondition of another query but not in the precondition or the postcondition of a command.
 - d. may appear in the class invariant.
3. The syntax of a program...
 - a. is the set of properties of its potential executions.
 - b. can be derived from the set of its objects.

- c. is the structure and the form of its text.
 - d. may be violated at run-time.
4. A class...
- a. is the description of a set of possible run-time objects to which the same features are applicable.
 - b. is a run-time entity from which objects can be created.
 - c. may be an implementation of an Abstract Data Type.
5. Void references...
- a. indicate missing information.
 - b. terminate loops.
 - c. indicate expanded objects.
 - d. terminate linked structures (e.g. lists).
6. Which of the following statements are true?
- a. A class invariant must hold after the execution of a creation procedure, then before and after the execution of any of the features of the class exported to clients.
 - b. Preconditions and postconditions cannot be instantiated, class invariants can.
 - c. A client calling a feature must make sure that the preconditions hold before the call.
 - d. A feature must make sure that, if its precondition held at the beginning of its execution, its postcondition will hold at the end. The class invariant does not have to hold at the end but it must hold during the execution of the feature.

2 Fairy tales

Goal

In this task you will have to categorize features, decide which expressions could be passed as arguments to specified features, and write down feature calls. The task uses the classes `FAIRY_TALE`, `FAIRY_TALE_FIGURE`, and `WRITER` (see listings 1, 2, and 3). Have a look at those classes now.

Listing 1: Class `FAIRY_TALE`

```
class
2  FAIRY_TALE
4 create
   make
```

```
6
feature -- Initialization
8
  make (a_name: STRING) is
10    -- Initialize 'name' with 'a_name'.
    require
12    a_name_not_void_and_empty: a_name /= Void and then not a_name.is_empty
    do
14    name := a_name
    ensure
16    name_initialized: name = a_name
    end
18
feature -- Access
20
  name: STRING
22    -- Name of fairy tale

  author: WRITER
24    -- Author of fairy tale
26

  protagonist: FAIRY_TALE_FIGURE
28    -- Main character of fairy tale

30 feature -- Element change

32 set_author (an_author: WRITER) is
    -- Set 'author' of fairy tale to 'an_author'.
    require
34    an_author_not_void: an_author /= Void
    do
36    author := an_author
    ensure
38    author_set: author = an_author
    end
40

42 set_protagonist (a_protagonist: FAIRY_TALE_FIGURE) is
    -- Set 'protagonist' of fairy tale to 'a_protagonist'.
    require
44    a_protagonist_not_void: a_protagonist /= Void
    do
46    protagonist := a_protagonist
    ensure
48    protagonist_set: protagonist = a_protagonist
    end
50

52 invariant
    name_exists: name /= Void and then not name.is_empty
54
end
```

Listing 2: Class *FAIRY_TALE_FIGURE*

```
class
2  FAIRY_TALE_FIGURE

4 create
  make
```

```
6
feature -- Initialization
8
  make (a_name: STRING) is
10    -- Initialize 'name' with 'a_name'.
    require
12    a_name_not_void_and_empty: a_name /= Void and then not a_name.is_empty
    do
14    name := a_name
    ensure
16    name_initialized: name = a_name
    is_good: is_good
18    end

20 feature -- Access

22 name: STRING
    -- Name of fairy tale figure

24 associated_fairy_tale : FAIRY_TALE
26    -- Fairy tale to which fairy tale figure belongs

28 enemy: FAIRY_TALE_FIGURE
    -- Enemy of fairy tale figure

30 friend: FAIRY_TALE_FIGURE
32    -- Friend of fairy tale figure.

34 feature -- Status report

36 is_good: BOOLEAN is
    -- Is fairy tale figure good?
38    do
    Result := not is_evil
40    ensure
    contrary_of_evil: is_good = not is_evil
42    end

44 is_evil: BOOLEAN
    -- Is fairy tale figure evil?

46 feature -- Status setting
48
    set_evil is
50    -- Make fairy tale figure evil.
    do
52    is_evil := True
    ensure
54    is_evil_set : is_evil
    end

56
    set_good is
58    -- Make fairy tale figure good.
    do
60    is_evil := False
    ensure
62    is_evil_set : is_good
```

```

    end
64
feature -- Element change
66
    set_associated_fairy_tale ( a_fairy_tale : FAIRY_TALE) is
68      -- Set 'associated_fairy_tale' of fairy tale figure to 'a_fairy_tale '.
        require
70        a_fairy_tale_not_void : a_fairy_tale /= Void
        do
72          associated_fairy_tale := a_fairy_tale
        ensure
74          associated_fairy_tale_set : associated_fairy_tale = a_fairy_tale
        end
76
    set_enemy (an_enemy: FAIRY_TALE_FIGURE) is
78      -- Set 'enemy' of fairy tale figure to 'an_enemy'.
        require
80        an_enemy_not_void: an_enemy /= Void
        do
82          enemy := an_enemy
        ensure
84          enemy_set: enemy = an_enemy
        end
86
    set_friend (a_friend: FAIRY_TALE_FIGURE) is
88      -- Set 'friend' of fairy tale figure to 'a_friend '.
        require
90        a_friend_not_void: a_friend /= Void
        do
92          friend := a_friend
        ensure
94          friend_set: friend = a_friend
        end
96
invariant
98  name_exists: name /= Void and then not name.is_empty
    either_good_or_evil : is_good = not is_evil
100
end
```

Listing 3: Class *WRITER*

```

class
2  WRITER

4 create
    make

6
feature -- Initialization
8
    make (a_name: STRING) is
10      -- Initialize 'name' with a_name.
        require
12        a_name_not_void_and_empty: a_name /= Void and then not a_name.is_empty
        do
14          name := a_name
        ensure
16          name_initialized: name = a_name
```

```

    end
18
feature -- Access
20
    name: STRING
22    -- Name of writer
24    most_famous_fairy_tale: FAIRY_TALE
26    -- Most famous fairy tale of writer
feature -- Element change
28
    set_most_famous_fairy_tale ( a_fairy_tale : FAIRY_TALE) is
30    -- Set 'most_famous_fairy_tale' of writer to ' a_fairy_tale '.
    require
32    a_fairy_tale_not_void : a_fairy_tale /= Void
    do
34    most_famous_fairy_tale := a_fairy_tale
    ensure
36    most_famous_fairy_tale_set: most_famous_fairy_tale = a_fairy_tale
    end
38
invariant
40    name_exists: name /= Void and then not name.is_empty
42 end
    
```

2.1 Feature categorization

Indicate the category of each feature in the table below by marking the respective columns with a cross. Please note that multiple crosses per row are possible.

feature	class	command	query	procedure	attribute	function
make	FAIRY_TALE					
name	FAIRY_TALE_FIGURE					
author	FAIRY_TALE					
set_most_famous_fairy_tale	WRITER					
is_evil	FAIRY_TALE_FIGURE					
set_protagonist	FAIRY_TALE					
is_good	FAIRY_TALE_FIGURE					

2.2 Typing

Indicate for each query in the table below to which of the listed features it could be passed as an argument. Mark the respective columns, if any, with a cross. The features in question are the following:

a set_enemy (an_enemy: FAIRY_TALE_FIGURE)

- b make (a_name: STRING)
- c set_most_famous_fairy_tale (a_fairy_tale: FAIRY_TALE)
- d set_friend (a_friend: FAIRY_TALE_FIGURE)

query	class	a	b	c	d
name	FAIRY_TALE_FIGURE				
is_good	FAIRY_TALE_FIGURE				
protagonist	FAIRY_TALE				
most_famous_fairy_tale	WRITER				
author	FAIRY_TALE				

2.3 Calling features

Complement the code sections below to achieve exactly what is asked.

2.3.1 Specifying the writer

Complement the feature `specify_writer` below with two feature calls using the classes given in listings 1, 2, and 3. The calls shall make the writer Grimm both the author of the fairy tales Rotkäppchen and Hänsel und Gretel.

Listing 4: Extract from *ROOT_CLASS*

```

specify_writer is
2  -- Specify writer.
   local
4   rotkaeppchen_maerchen: FAIRY_TALE
   haensel_und_gretel_maerchen: FAIRY_TALE
6   grimm: WRITER
   do
8   create rotkaeppchen_maerchen.make ("Rotkaeppchen")
   create haensel_und_gretel_maerchen.make ("Haensel und Gretel")
10  create grimm.make ("Grimm")
   .....
12  .....
   end
    
```

2.3.2 Making friends

Complement the feature `make_friends` below with two feature calls using the classes given in listings 1, 2, and 3. The first call shall make the fairy tale figure `rotkaeppchen` become the protagonist of the Rotkäppchen fairy tale and the second shall make the protagonist of Grimm’s most famous fairy tale become the friend of the fairy tale figure `grossmutter`. Do not use the entity `rotkaeppchen` in the second feature call.

Listing 5: Extract from *ROOT_CLASS*

```

make_friends is
2  -- Make friends.
    
```

```
local
4  grimm: WRITER
   rotkaeppchen_maerchen: FAIRY_TALE
6  rotkaeppchen: FAIRY_TALE-FIGURE
   grossmutter: FAIRY_TALE-FIGURE
8  do
   create grimm.make ("Grimm")
10 create rotkaeppchen_maerchen.make ("Rotkaeppchen")
   create rotkaeppchen.make ("Rotkaeppchen")
12 create grossmutter.make ("Grossmutter")
   grimm.set_most_famous_fairy_tale (rotkaeppchen_maerchen)
14 .....
   .....
16 end
```

3 Doubly linked lists

Goal

In this task you have to implement a data structure called doubly linked list. The structure consists of two classes: `INTEGER_LIST_CELL` and `INTEGER_LIST`. An object of type `INTEGER_LIST_CELL` holds an `INTEGER` as the cell content and has a `previous` and a `next` reference to another object of type `INTEGER_LIST_CELL`. By attaching the `previous` and `next` references correctly two or more cells can be connected to form a list. The class `INTEGER_LIST` offers functionality to access the first and the last cell of a list, and to add a new cell to the end. In figure 1 you see a drawing of a doubly linked list.

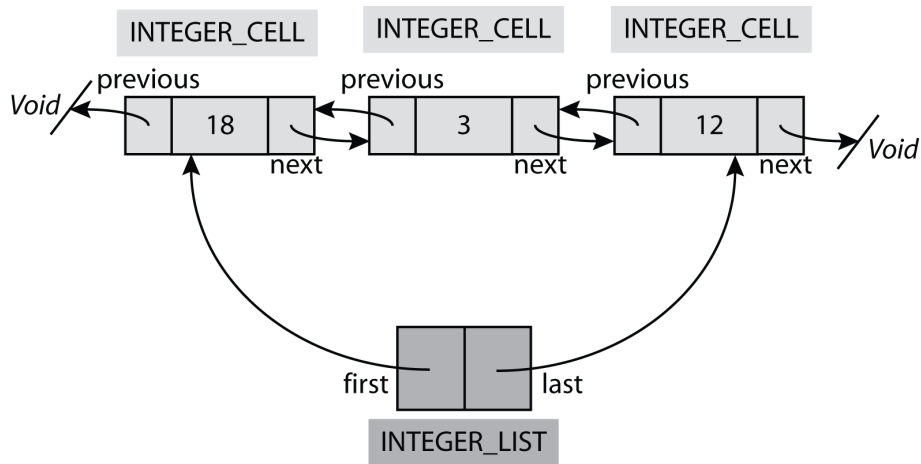


Figure 1: Doubly linked list

3.1 Class `INTEGER_LIST_CELL`

Read through the class `INTEGER_LIST_CELL` in listing 6. You will need the features of this class for the rest of the task.

3.2 Class `INTEGER_LIST`

1. Implement the feature `extend` of class `INTEGER_LIST` (see listing 7). This feature takes an `INTEGER` as argument, generates a new object of type `INTEGER_LIST_CELL` with the according content and puts the new cell at the end of the list. Make sure that your implementation satisfies the given postconditions of the feature.
2. Implement the feature `search` of class `INTEGER_LIST` (see listing 7). This feature goes through the list and looks for a cell with the same content as the given argument. If it is found the found element is made available through the feature `found_cell`. Note the postcondition!

The dotted lines provide enough space for your solution, so you can fill the code directly into the sheet.

Listing 7: Class `INTEGER_LIST`

```
class
2  INTEGER_LIST

4 create
   make_empty

6 feature -- Initialization
8
   make_empty is
10  -- Initialize the list to be empty.
   do
12    first := void
    last := void
14    count := 0
   end

16 feature -- Access
18
   first : INTEGER_LIST_CELL
20   -- Head element of the list, Void if the list is empty

22  last : INTEGER_LIST_CELL
   -- Tail element of the list , Void if the list is empty

24  is_found : BOOLEAN
26   -- Was the element that was last searched for found?

28  found_cell : INTEGER_LIST_CELL
   -- Found cell of last search (may be Void!)

30 feature -- Element change
32
```

```
34 extend (a_value: INTEGER) is
    -- Append a integer list cell with content 'a_value' at the end of the list .
    local
36     el: INTEGER_LIST_CELL
    do
38     .....
40     .....
42     .....
44     .....
46     .....
48     .....
50     .....
52     .....
54     .....
56     .....
58     .....
60     .....
62     .....
64     .....
66     .....
68     .....
70     .....
72     .....
74 ensure
    one_more: count = old count + 1
76     first_set : count = 1 implies first.value = a_value
    last_set : last.value = a_value
78 end

80 feature -- Search
82 search (a_value: INTEGER) is
    -- Search for the first element with content 'a_value'.
84     local
        el: INTEGER_LIST_CELL
86     do
88     .....
```

```
90
92 .....
94 .....
96 .....
98 .....
100 .....
102 .....
104 .....
106 .....
108 .....
110 .....
112 .....
114 .....
116 .....
118 .....
120 .....
122
124   ensure
      found: (is_found and then found_cell /= Void) or else (not is_found and found_cell
          = Void)
126   end
128   feature -- Measurement
130     count: INTEGER
        -- Number of cells in the list
132   feature -- Status report
134     empty: BOOLEAN is
        -- Is the list empty?
136     do
        Result := (count = 0)
138     end
140   end
```

Listing 6: Class *INTEGER_LIST_CELL*

```
class
2  INTEGER_LIST_CELL

4  create
   set_value

6
   feature -- Access
8
   value: INTEGER
10      -- Content that is stored in the list cell

12  next: INTEGER_LIST_CELL
      -- Reference to the next integer list cell of a list

14  previous: INTEGER_LIST_CELL
16      -- Reference to the previous integer list cell of a list

18  feature -- Element change

20  set_value (x: INTEGER) is
      -- Set 'value' to 'x'.
22      do
         value := x
24      ensure
         value_set: value = x
26      end

28  set_next (el: INTEGER_LIST_CELL) is
      -- Set next to 'el'.
30      do
         next := el
32      ensure
         next_set: next = el
34      end

36  set_previous (el: INTEGER_LIST_CELL) is
      -- Set previous to 'el'.
38      do
         previous := el
40      ensure
         previous_set: previous = el
42      end

44  end
```