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.
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

```java
class FAIRY_TALE {
    create
    make
}
```
feature -- Initialization

make (a_name: STRING) is
   -- Initialize 'name' with 'a_name'.
   require
   a_name_not_void_and_empty: a_name /= Void and then not a_name.is_empty
   do
      name := a_name
   ensure
      name_initialized: name = a_name
   end

feature -- Access

name: STRING
   -- Name of fairy tale

author: WRITER
   -- Author of fairy tale

protagonist: FAIRY_TALE FIGURE
   -- Main character of fairy tale

feature -- Element change

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

set_protagonist (a_protagonist: FAIRY_TALE FIGURE) is
   -- Set 'protagonist' of fairy tale to 'a_protagonist'.
   require
   a_protagonist_not_void: a_protagonist /= Void
   do
      protagonist := a_protagonist
   ensure
      protagonist_set: protagonist = a_protagonist
   end

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

Listing 2: Class FAIRY_TALE FIGURE

class
   FAIRY_TALE FIGURE
create
   make
feature -- Initialization

make (a_name: STRING) is
  -- Initialize 'name' with 'a_name'.
require
  a_name_not_void_and_empty: a_name /= Void and then not a_name.is_empty
do
  name := a_name
ensure
  name_initialized: name = a_name
  is_good: is_good
end

feature -- Access

name: STRING
  -- Name of fairy tale figure
associated_fairy_tale: FAIRY_TALE
  -- Fairy tale to which fairy tale figure belongs
enemy: FAIRY_TALE,FIGURE
  -- Enemy of fairy tale figure
friend: FAIRY_TALE,FIGURE
  -- Friend of fairy tale figure.

feature -- Status report

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

is_evil: BOOLEAN
  -- Is fairy tale figure evil?

feature -- Status setting

set_evil is
  -- Make fairy tale figure evil.
do
  is_evil := True
ensure
  is_evil_set: is_evil
end

set_good is
  -- Make fairy tale figure good.
do
  is_evil := False
ensure
  is_evil_set: is_good
Listing 3: Class WRITER

class
2 WRITER
create
make

feature -- Initialization
8
make (a_name: STRING) is
-- 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
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.

<table>
<thead>
<tr>
<th>feature</th>
<th>class</th>
<th>command</th>
<th>query</th>
<th>procedure</th>
<th>attribute</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>FAIRY_TALE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>FAIRY_TALE_FIGURE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>author</td>
<td>FAIRY_TALE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set_most_famous_fairy_tale</td>
<td>WRITER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_evil</td>
<td>FAIRY_TALE_FIGURE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set_protagonist</td>
<td>FAIRY_TALE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_good</td>
<td>FAIRY_TALE_FIGURE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>query</th>
<th>class</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>FAIRY_TALE FIGURE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_good</td>
<td>FAIRY_TALE FIGURE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>protagonist</td>
<td>FAIRY_TALE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>most_famous_fairy_tale</td>
<td>WRITER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>author</td>
<td>FAIRY_TALE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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

```java
specify_writer is
  -- Specify writer.
  local
    rotkaeppchen_maeerchen: FAIRY_TALE
    haensel_und_gretel_maeerchen: FAIRY_TALE
    grimm: WRITER
  do
    create rotkaeppchen_maeerchen.make("Rotkaeppchen")
    create haensel_und_gretel_maeerchen.make("Haensel und Gretel")
    create grimm.make("Grimm")
  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 rotkaepchen 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 rotkaepchen in the second feature call.

Listing 5: Extract from ROOT_CLASS

```java
make_friends is
  -- Make friends.
```
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.

![Doubly linked list diagram]

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 INTEGER_LIST

create
    make_empty

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

feature -- Access

    first : INTEGER_LIST_CELL
        -- Head element of the list, Void if the list is empty

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

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

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

feature -- Element change
extend (a_value: INTEGER) is
    -- Append a integer list cell with content ‘a_value’ at the end of the list.
local
el: INTEGER_LIST_CELL
do
    ensure
        one_more: count = old count + 1
        first_set : count = 1 implies first.value = a_value
        last_set : last.value = a_value
end

feature -- Search

search (a_value: INTEGER) is
    -- Search for the first element with content ‘a_value’.
local
el: INTEGER_LIST_CELL
do
ensure
  found: (is_found and then found.cell /= Void) or else (not is_found and found.cell = Void)
end

feature -- Measurement

  count: INTEGER  
  -- Number of cells in the list

feature -- Status report

  empty: BOOLEAN is
  -- Is the list empty?
  do
    Result := (count = 0)
  end
end
Listing 6: Class INTEGER_LIST_CELL

class INTEGER_LIST_CELL

create
set_value

feature -- Access
value: INTEGER
  -- Content that is stored in the list cell
next: INTEGER_LIST_CELL
  -- Reference to the next integer list cell of a list
previous: INTEGER_LIST_CELL
  -- Reference to the previous integer list cell of a list

feature -- Element change
set_value (x: INTEGER) is
  -- Set 'value' to 'x'.
do
  value := x
ensure
  value_set: value = x
end

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

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

end