

Mock Exam 2

ETH Zurich

Date: 7./8. December 2009

Name: _____

Group: _____

Directions:

- Exam duration: 90 minutes.
- Use a pen (**not** a pencil)!
- Only one solution can be handed in per question. Invalid solutions need to be crossed out clearly.
- Please write legibly! We will only correct solutions that we can read.
- Manage your time carefully (take into account the number of points for each question).
- You get **bonus points** for correct loop variants and loop invariants.
- Please **immediately** tell the exam supervisors if you feel disturbed during the exam.

feature -- Status report

color: *CHARACTER*
-- The card color

value: *INTEGER*
-- The card value

is_valid_color (*c*: *CHARACTER*): *BOOLEAN*
-- Is 'c' a valid color?

require

.....
.....
.....

ensure

.....
.....
.....

end

is_in_range (*n*: *INTEGER*): *BOOLEAN*
-- Is 'n' in the acceptable range of values?

require

.....
.....
.....

ensure

.....
.....
.....

end

invariant

.....
.....
.....

end

```
class
  DECK

create
  make

feature -- Creation

  make
    -- Create deck.
  require
    .....
    .....
    .....
  ensure
    .....
    .....
    .....
  end

feature -- Status report

  is_empty: BOOLEAN
    -- Is this deck empty?
  do
    Result := card_list.is_empty
  end

  count: INTEGER
    -- Number of remaining cards in deck.
  do
    Result := card_list.count
  end

feature -- Access

  top_card: CARD
    -- Top card of deck.
```

feature *-- Removal*

remove_top_card
-- Remove top card from deck.

require

.....
.....
.....

ensure

.....
.....
.....

end

feature {*NONE*} *-- Implementation*

card_list: *LINKED_LIST* [*CARD*]
-- Implementation of the card list

invariant

.....
.....
.....

end

2 Media (7 points)

2.1 Background Information

Software used by a media shop models books, magazines, DVDs and electronic books. For each of these media types there is a corresponding class. Books, magazines and electronic books can be printed out on paper. Thus each of the corresponding classes offers a command *print_out* with a specific implementation. A DVD can not be printed out on paper, but it can be played. Thus the DVD class offers a command *play* with a specific implementation. An electronic book is a book which is available in a digital format. Such an electronic book is used in conjunction with a reader device, which can play the electronic book. However, an electronic book can also be printed on paper, if necessary. Therefore the electronic book class offers the two commands *print_out* and *play* providing a specific implementation. Every medium has a name. Thus each of the classes offers an attribute *name* of type *STRING*. Figure ?? shows the class diagram of these classes.

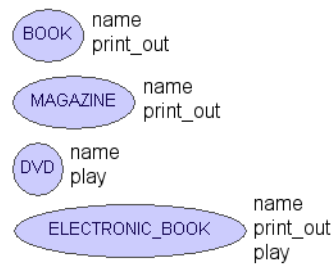


Figure 1: class diagram showing the initial situation

2.2 Task

The problem with the class diagram in figure ?? is the lack of abstractions. It is not possible to abstractly look at multiple media sharing common features. Your task is to re-engineer the class diagram by abstracting common features in parent classes using inheritance. Draw a new class diagram showing the classes, the features and the inheritance arrows. Don't forget to mention whether a class is deferred or effective and whether a feature is deferred, effective or redefined. YOU DO NOT NEED TO PROVIDE ANY CODE.

3 Tree Iteration (12 Points)

The following class `TREE [G]` represents n-ary trees. A tree consists of a root node, which can have an arbitrarily many children nodes. Each child node itself can have arbitrary many children. In fact each child node itself is a tree, with itself as a root node.

```
class TREE [G]

  create
    make

  feature {NONE} -- Initialization

    make (v: G) is
      -- Create new cell with value 'v'.
      require
        v_not_void: v /= Void
      do
        value := v
        create {LINKED_LIST [TREE [G]]} children.make
      ensure
        value_set: value = v
      end

  feature -- Access

    value: G
      -- Value of node

    children: LIST [TREE [G]]
      -- Child nodes of this node

  feature -- Insertion

    put (v: G) is
      -- Add child cell with value 'v' as last child.
      require
        v_not_void: v /= Void
      local
        c: TREE [G]
      do
        create c.make (v)
        children.extend (c)
      ensure
        one_mode: children.count = old children.count + 1
        inserted: children.last.value = v
      end

  invariant
    children_not_void: children /= Void
    value_not_void: value /= Void
```

end

The following gives relevant aspects of the interface of class *LIST* [*G*]. Class *LINKED_LIST* [*G*] is a descendant of class *LIST* [*G*].

deferred class interface *LIST* [*G*]

feature -- Access

index: *INTEGER*
-- Index of current position.

item: *G*
-- Item at current position.

require
not_off: **not** *off*

feature -- Measurement

count: *INTEGER*
-- Number of items.

feature -- Status report

after: *BOOLEAN*
-- Is there no valid cursor position to the right of cursor?

before: *BOOLEAN*
-- Is there no valid cursor position to the left of cursor?

off: *BOOLEAN*
-- Is there no current item?

is_empty: *BOOLEAN is*
-- Is structure empty?

feature -- Cursor movement

back
-- Move to previous position.
require
not_before: **not** *before*
ensure
moved_back: *index* = **old** *index* - 1

finish
-- Move cursor to last position.
-- (No effect if empty)
ensure
not_before: **not** *is_empty* **implies not** *before*

forth
-- Move to next position.

```
require
  not_after: not after
ensure
  moved_forth: index = old index + 1

start
  -- Move cursor to first position.
  -- (No effect if empty)
ensure
  not_after: not is_empty implies not after

feature -- Element change

extend (v: G)
  -- Add a new occurrence of 'v'.
ensure
  one_more: count = old count + 1

invariant
  before_definition: before = (index = 0)
  after_definition: after = (index = count + 1)
  non_negative_index: index >= 0
  index_small_enough: index <= count + 1
  off_definition: off = ((index = 0) or (index = count + 1))
  not_both: not (after and before)
  before_constraint: before implies off
  after_constraint: after implies off
  empty_definition: is_empty = (count = 0)
  non_negative_count: count >= 0
end
```

3.1 Traversing the tree

Class *ROOT_CLASS* below first builds a tree and then prints the values of the tree in two different ways: pre-order and post-order.

Fill in the missing source code of class *ROOT_CLASS* so that its *make* feature prints the following:

```
1
1.1
1.1.1
1.1.2
1.2
1.3
1.3.1
---
1.1.1
1.1.2
1.1
1.2
1.3.1
1.3
1
```

```
class
  ROOT_CLASS

create
  make

feature

  make is
    -- Run program.
    local
      root: TREE [STRING]
      cell: TREE [STRING]
    do
      create root.make ("1")
      root.put ("1.1")
      cell := root.children.last
      cell.put ("1.1.1")
      cell.put ("1.1.2")
      root.put ("1.2")
      root.put ("1.3")
      cell := root.children.last
      cell.put ("1.3.1")

      print_pre_order (root)
      io.put_string ("---")
      io.put_new_line
      print_post_order (root)
    end
```

```
print_pre_order (t: TREE [STRING]) is  
    -- Print tree in pre-order.
```

```
require
```

```
    t_not_void: t /= Void
```

```
local
```

```
.....
```

```
.....
```

```
.....
```

```
do
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
.....
```

```
end
```

```
print_post_order (t: TREE [STRING]) is  
    -- Print tree in post-order.
```

```
    require  
        t_not_void: t /= Void
```

```
    local
```

```
        .....
```

```
        .....
```

```
        .....
```

```
    do
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
        .....
```

```
    end
```

```
end
```

4 Integration of an Integration (12 Points)

Consider the following simplified interface of class *FUNCTION*.

interface class

FUNCTION [*BASE_TYPE*, *OPEN_ARGS* → *TUPLE*, *RESULT_TYPE*]

feature -- Access

item (*args*: *OPEN_ARGS*): *RESULT_TYPE*

-- Result of calling function with 'args' as operands.

end

Fill in the body of routine *integrate* (6 Points) and *integrate_2d* (6 Points) in the class *INTEGRATOR* below in such a way that:

- *integrate* sums up the results of the supplied function from 'lower' to 'upper'.
- If 'upper' is smaller than 'lower', the result is zero.
- *integrate_2d* does the same thing for functions with 2 integer inputs. It sums up the results of the supplied function in the whole 2 dimensional rectangle defined by 'l_x' to 'u_x' and 'l_y' to 'u_y'.
- If the area of the rectangle is empty (either because 'u_x' is smaller than 'l_x' or 'u_y' is smaller than 'l_y'), the result is zero.
- In order to implement *integrate_2d* you **must** make use of routine *integrate*. One way to do this is by involving the helper-routine *apply* from class *INTEGRATOR*.

class

INTEGRATOR

feature

integrate (*f*: *FUNCTION* [*ANY*, *TUPLE* [*INTEGER*], *INTEGER*];
lower, *upper*: *INTEGER*): *INTEGER* **is**

require

f_not_void: *f* /= **Void**

local

.....

.....

.....

do

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

ensure

empty_interval: *upper* < *lower* **implies** **Result** = 0

end

