

# Mock Exam 2 Solution

ETH Zurich

## 1 Design by Contract (9 Points)

Classes *CARD* and *DECK* are part of a software system that models a card game. The following is an extract from the game rules booklet:

1. A deck is initially made of 36 cards.
2. Every card represents a value in the range 2..10. Furthermore, every card represents one color out of four possible colors.
3. The colors represented in the game cards are red ('R'), white ('W'), green ('G') and blue ('B').
4. As long as there are cards in the deck, the players can look at the top card and remove it.

Your task is to fill in the contracts of the two classes *CARD* and *DECK* (preconditions, postconditions and class invariants), according to the specification given. You are not allowed to change the interfaces of the classes or any of the already given implementations. Note that the number of dotted lines does not indicate the number of code lines that you have to provide, or if you have to provide a contract at all.

### 1.1 Solution

**class**

*CARD*

**create**

*make*

**feature** -- Creation

```
make (a_color: CHARACTER; a_value: INTEGER)
  -- Create a card given a color and a value.
require
  is_valid_color (a_color)
  is_in_range (a_value)
do
  color := a_color
  value := a_value
ensure
  color_set: color = a_color
  value_set: value = a_value
end
```

**feature** -- Status report

*color*: *CHARACTER*  
-- The card color

*value*: *INTEGER*  
-- The card value

*is\_valid\_color* (*c*: *CHARACTER*): *BOOLEAN*  
-- Is 'c' a valid color?  
**do**  
  **Result** := (*c* = 'R' **or** *c* = 'B' **or** *c* = 'W' **or** *c* = 'G')  
**ensure**  
  **Result** = (*c* = 'R' **or** *c* = 'B' **or** *c* = 'W' **or** *c* = 'G')  
**end**

*is\_in\_range* (*n*: *INTEGER*): *BOOLEAN*  
-- Is 'n' in the acceptable range of values?  
**do**  
  **Result** := (2 <= *n* **and** *n* <= 10)  
**ensure**  
  **Result** = (2 <= *n* **and** *n* <= 10)  
**end**

**invariant**

*valid\_color*: *is\_valid\_color* (*color*)  
*valid\_range*: *is\_in\_range* (*value*)

**end**

**class**

*DECK*

**create**

*make*

**feature** -- Creation

*make*  
-- Create deck.  
**do**  
  ...  
**ensure**  
  *deck\_filled*: *count* = 36  
**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
        not_empty: not is_empty
    do
        card_list.start
        card_list.remove
        if card_list.is_empty then
            top_card := Void
        else
            top_card := card_list.item
        end
    ensure
        one_card_less_in_deck: count = old count - 1
        top_card_replaced: top_card /= old top_card
    end

feature {NONE} -- Implementation

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

invariant
    is_legal_deck: 0 <= count and count <= 36
    top_card_available: is_empty = (top_card = Void)
    count_empty_relation: is_empty = (count = 0)
    card_list_exists: card_list /= Void
    count_corresponds: count = card_list.count
    top_card_is_first: not is_empty implies top_card = card_list.first

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.

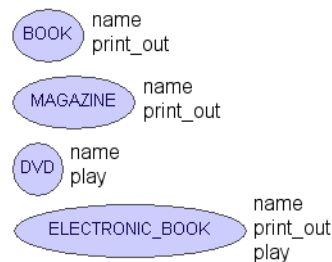


Figure 1: class diagram showing the initial situation

### 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.

### 2.3 Solution

See Figure ??.

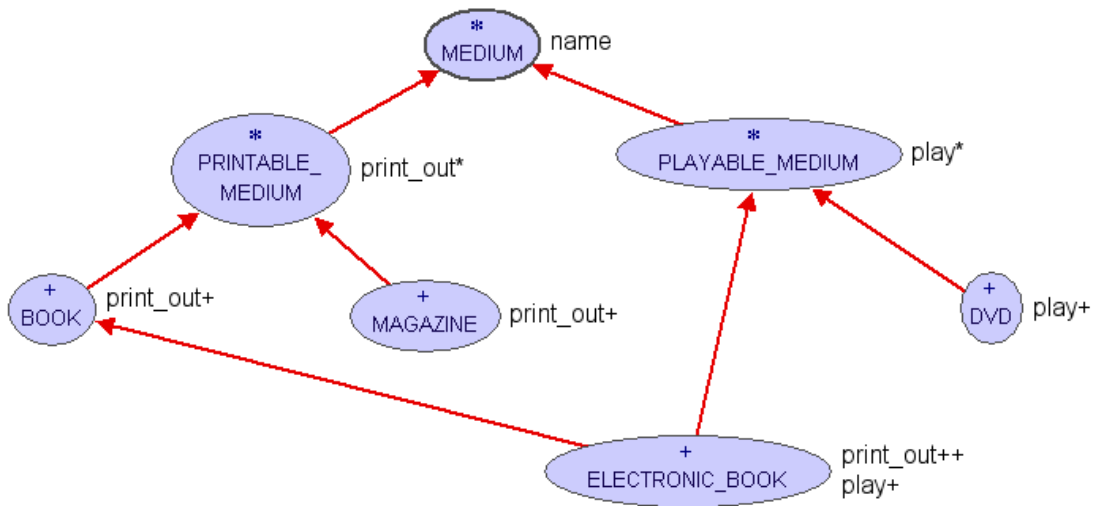


Figure 2: media solution class diagram

### 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
```

### 3.2 Solution

```
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
    do
        io.put_string (t.value)
        io.put_new_line
        from
            t.children.start
        until
            t.children.off
        loop
            print_pre_order (t.children.item)
            t.children.forth
        variant
            t.children.count - t.children.index + 1
        end
    end
end

print_post_order (t: TREE [STRING]) is
    -- Print tree in post-order.
    require
        t_not_void: t /= Void
    do
        from
            t.children.start
        until
            t.children.off
        loop
            print_post_order (t.children.item)
            t.children.forth
        variant
            t.children.count - t.children.index + 1
        end
        io.put_string (t.value)
        io.put_new_line
    end
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*.

### 4.1 Solution

#### 4.1.1 Loop version

**class** *INTEGRATOR*

**feature**

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

**require**

*f\_not\_void*: *f* /= **Void**

**local**

*x*: *INTEGER*

**do**

**from**

*x* := *lower*

**invariant**

**Result** = *integrate* (*f*, *lower*, *x* - 1)

**until**

*x* > *upper*

**loop**

**Result** := **Result** + *f.item* ([*x*])

*x* := *x* + 1

```
    variant
      upper - x + 1
    end
  ensure
    empty_interval: upper < lower implies Result = 0
  end

integrate_2d (f: FUNCTION [ANY, TUPLE [INTEGER, INTEGER], INTEGER];
             l_x, l_y, u_x, u_y: INTEGER): INTEGER is
  require
    f_not_void: f /= Void
  local
    x: INTEGER
  do
    from
      x := l_x
    invariant
      Result = integrate_2d (f, l_x, l_y, x - 1, u_y)
    until
      x > u_x
    loop
      Result := Result + integrate (agent apply (f, x, ?), l_y, u_y)
      x := x + 1
    variant
      u_x - x + 1
    end
  ensure
    empty_interval: (u_x < l_x) or (u_y < l_y) implies Result = 0
  end

feature {NONE} -- Implementation

  apply (f: FUNCTION [ANY, TUPLE [INTEGER, INTEGER], INTEGER];
        x, y: INTEGER): INTEGER is
    do
      Result := f.item ([x, y])
    end

end
```

#### 4.1.2 Recursive version

```
class INTEGRATOR
```

```
feature
```

```
  integrate (f: FUNCTION [ANY, TUPLE [INTEGER], INTEGER];
            lower, upper: INTEGER): INTEGER is
    require
      f_not_void: f /= Void
    do
      if upper >= lower then
```

```

    Result := f.item ([lower]) + integrate (f, lower + 1, upper)
  end
ensure
  empty_interval: upper < lower implies Result = 0
end

integrate_2d (f: FUNCTION [ANY, TUPLE [INTEGER, INTEGER], INTEGER];
              l_x, l_y, u_x, u_y: INTEGER): INTEGER is
  require
    f_not_void: f /= Void
  do
    if upper >= lower then
      Result := integrate (agent apply (f, l_x, ?), l_y, u_y) +
                integrate_2d (f, l_x + 1, l_y, u_x, u_y)
    end
  ensure
    empty_interval: (u_x < l_x) or (u_y < l_y) implies Result = 0
  end

feature {NONE} -- Implementation

  apply (f: FUNCTION [ANY, TUPLE [INTEGER, INTEGER], INTEGER];
         x, y: INTEGER): INTEGER is
    do
      Result := f.item ([x, y])
    end

end
```