Software Architecture Exam

Summer Semester 2007
Prof. Dr. Bertrand Meyer
Date: 19 June 2007

Family name, first name: ....

Student number: ....

I confirm with my signature, that I was able to take this exam under regular circumstances and that I have read and understood the directions below.

Signature: ....

Directions:

- Exam duration: 90 minutes.
- Except for a dictionary you are not allowed to use any supplementary material.
- Use a pen (**not** a pencil)!
- Please write your student number onto **each** sheet.
- All solutions can be written directly onto the exam sheets. If you need more space for your solution ask the supervisors for a sheet of official paper. You are **not** allowed to use other paper.
- 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).
- Don't forget to add comments to features.
- Please **immediately** tell the supervisors of the exam if you feel disturbed during the exam.

Good luck!

Question	Number of possible points	Points
1	10	
2	10	
3	11	
4	18	
5	17	

1 Abstract Data Types (10 Points)

The MyMusic shop sells music CDs. The shop needs to keep track of the CD titles they have, so for each CD title they need to know:

- the name of the artist
- the title of the album
- the price
- how many copies they have on stock

It should also be possible to set a different price for a CD than it was created with, to sell CDs (no more than there are on stock), and to order new copies of a certain CD when there are none left on stock.

The following ADT should model this notion. Note that although it is called "CD" it represents a CD title, not an individual CD (e.g. "Mozart's 4oth Symphony, recorded by Karajan and published by Deutsche Gramophon", not one particular CD with that title). Types STRING and INTEGER are considered given with the usual semantics and are opaque types (this means their own properties are not visible and do not matter in the exercise).

TYPES

CD, STRING, INTEGER

FUNCTIONS

new_cd: STRING $\times$ STRING $\times$ INTEGER $\times$ INTEGER $\rightarrow$ CD

title: CD $\rightarrow$ STRING

artist: CD $\rightarrow$ STRING

price: CD $\rightarrow$ INTEGER

quantity: CD $\rightarrow$ INTEGER

set_price: CD $\times$ INTEGER $\rightarrow$ CD

sell: CD $\times$ INTEGER $\rightarrow$ CD

order_new: CD $\times$ INTEGER $\rightarrow$ CD

The informal semantics of these functions is the following:

- "new_cd" yields a new CD with the data it receives as argument (in this order): title, artist, price, quantity (the initial quantity on stock)
- "title", "artist", "price", "quantity" return the corresponding characteristics of a CD
- "set_price" sets the price of a CD to the given argument
- "sell" reduces the quantity of CDs on stock by the given number
- "order_new" increases the quantity of CDs on stock with the given number

The business model of the shop imposes the following constraints:

- The quantity on stock must always be non-negative.
- New copies of a CD can only be ordered when the shop does not have it on stock anymore.
- The price of a CD must be strictly greater than 0.
- A new CD can only be created if there is at least one copy on stock.

To Do:

1. Which functions from the above list are (1 POINT):
   (a) Creators: ....
   (b) Queries: ....
   (c) Commands: ....

   (you only need to specify the functions' names)
2. Mark the functions that should be partial in the **FUNCTIONS** section (by crossing the arrow in the function definition) (1 POINT).
3. Write the **PRECONDITIONS** section of the CD ADT (4 POINTS).

   ....

   ....

   ....

   ....

   ....

   ....

   ....

   ....

   ....

   ....

   ....

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

4. Write the **AXIOMS** section of the CD ADT so that this ADT is sufficiently complete (you don't need to prove sufficient completeness) (4 POINTS).

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

2 Design by Contract (10 Points)

The class *COUNTER* represents a natural counter with routines *increment* and *decrement*. The counter is implemented as an *INTEGER*. The class *STUDENT* represents students that take courses. The class *COURSE* represents courses. A course consists of a name, a list of students that are registered in the course, and a counter that stores the number of students registered in the course. In the following classes implementing this notion, complete the contracts at the locations marked by dotted lines. The first contract (the postcondition of feature *make* of class *COUNTER*) is done as an example. Part of the interface of class *LINKED_LIST* is provided to help the development of the contracts of class *COURSE*.

```
indexing
    description: "Objects that represent a natural counter."

class
    COUNTER
create
    make

feature -- Initialization

    make is
            -- Create a counter initializing it with 0.
        do
            item := 0
        ensure
            initial_value_is_0 : item = 0
        end

feature -- Element change
    increment is
            -- Increment the counter by 1.
        do
            item := item + 1
        ensure


            ..........................................................................
        end

    decrement is
            -- Decrement the counter by 1.
        require


            ..........................................................................
        do
            item := item - 1
        ensure


            ..........................................................................
        end

feature -- Implementation
    item: INTEGER

invariant


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

end
```

```
indexing
    description: "Objects that represent students. A student consists of a name."
class
    STUDENT

create
    make

feature -- Initialization
    make (n: STRING) is
```

```
            -- Create a student whose name is ‘n’.
        require

            ..........................................................................
        do
            name := n
        ensure

            ..........................................................................
        end

feature -- Access

    set_name (n: STRING) is
            -- Set the name.
        require

            ..........................................................................
        do
            name := n
        ensure

            ..........................................................................
        end


feature -- Implementation
    name: STRING

invariant

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

end
```

```
indexing
    description: "Objects that represent courses"

class
    COURSE

create
    make

feature {NONE} -- Initialization

    make (n: STRING) is
            -- Create a new course with name ’n’.
        require

            ..........................................................................
        do
            name := n
            create count_students.make
            create students.make
        ensure
```

```
            ..........................................................................
        end


feature -- Basic operations
    register (s: STUDENT) is
            -- Register a student.
        require

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

            ..........................................................................
        do
            students.extend (s)
            count_students.increment
        ensure

            ..........................................................................
        end

    delete (s: STUDENT) is
            -- Delete a student from the course.
        require

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

            ..........................................................................
        do
            students.start
            students.prune (s)
            count_students.decrement
        ensure

            ..........................................................................
        end

feature -- Implementation
    name: STRING
    count_students: COUNTER
    students: LINKED_LIST [STUDENT]

invariant

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

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

            ..........................................................................
end

class interface
```

```
    LINKED_LIST [G]

create
    make

feature -- Access

    cursor: LINKED_LIST_CURSOR [G]
            -- Current cursor position

    first : like item
            -- Item at first position

    index: INTEGER_32
            -- Index of current position

    item: G
            -- Current item

    last: like item
            -- Item at last position

feature -- Measurement

    count: INTEGER_32
            -- 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?

    full : BOOLEAN is False
            -- Is structured filled to capacity? (Answer: no.)

    is_inserted (v: G): BOOLEAN
            -- Has 'v' been inserted at the end by the most recent put or
            -- extend?

    has (v: like item): BOOLEAN is
        -- Does linked list include 'v'?

    isfirst : BOOLEAN
            -- Is cursor at first position?

    islast : BOOLEAN
            -- Is cursor at last position?

    off: BOOLEAN
            -- Is there no current item?

    readable: BOOLEAN
            -- Is there a current item that may be read?

    valid_cursor (p: CURSOR): BOOLEAN
            -- Can the cursor be moved to position 'p'?

feature -- Cursor movement
```

```
    back
            -- Move to previous item.

    finish
            -- Move cursor to last position.
            -- (Go before if empty)

    forth
            -- Move cursor to next position.

    go_to (p: CURSOR)
            -- Move cursor to position 'p'.

    start
            -- Move cursor to first position.

    search (v: like item) is
        -- Move to first position (at or after current
        -- position) where 'item' and 'v' are equal.
        -- If structure does not include 'v' ensure that
        -- 'exhausted' will be true.

feature -- Element change

    extend (v: like item)
            -- Add 'v' to end.
            -- Do not move cursor.

    merge_left (other: like Current)
            -- Merge 'other' into current structure before cursor
            -- position. Do not move cursor. Empty 'other'.

    merge_right (other: like Current)
            -- Merge 'other' into current structure after cursor
            -- position. Do not move cursor. Empty 'other'.

    put_front (v: like item)
            -- Add 'v' to beginning.
            -- Do not move cursor.

    replace (v: like item)
            -- Replace current item by 'v'.

feature -- Removal

    remove
            -- Remove current item.
            -- Move cursor to right neighbor
            -- (or after if no right neighbor).

    prune (v: like item) is
            -- Remove first occurrence of 'v', if any,
            -- after cursor position.
            -- If found, move cursor to right neighbor;
            -- if not, make structure 'exhausted'.

end -- class LINKED_LIST
```

3 Design Pattern Categories (11 Points)

Design patterns can be classified in terms of the underlying problem they are solving. In the lecture, you have seen three categories of design patterns: *creational design patterns*, *structural design patterns*, and *behavioral design patterns*. Assign each of the design patterns below to one of these three categories by writing its name into the according list. For each of the three categories choose one pattern and describe it in one or two sentences.

List: Composite, State, Abstract Factory, Singleton, Chain of Responsibility, Builder, Bridge, Strategy, Decorator, Flyweight

---

Example

List: Memento, Iterator, Interpreter

1. Behavioral design patterns:

    - Name: *Iterator*

      Description: *The iterator pattern provides a mechanism that allows sequential access to the elements of an aggregate object without exposing its underlying representation. In the iterator pattern each effective representation of the aggregate object has a corresponding effective iterator that provides operations* ***start****,* ***forth****,* ***off****, and* ***item****.*
    - Name: *Interpreter*
    - Name: *Memento*

Fill in here:

Each correctly categorized pattern is worth 0.5 Point. For each correct pattern description you get 2 Points.

1. Creational design patterns:

    - Name: ....

      Description: ....

      ....

      ....

      ....

      ....

. . . .

- Name: . . . .
- Name: . . . .
- Name: . . . .
- Name: . . . .
- Name: . . . .

2. Behavioral design patterns:
   - Name: . . . .

     Description: . . . .

     . . . .

     . . . .

     . . . .

     . . . .

     . . . .
   - Name: . . . .
   - Name: . . . .
   - Name: . . . .
   - Name: . . . .
   - Name: . . . .

3. Structural design patterns:
   - Name: . . . .

     Description: . . . .

     . . . .

. . . .

. . . .

. . . .

. . . .

- Name: . . . .
- Name: . . . .
- Name: . . . .
- Name: . . . .
- Name: . . . .

4 Observer (18 Points)

Below you will find a possible implementation for an application using the Observer design pattern:

```
deferred class
  OBSERVER
feature -- Basic operations
  update (a_subject: SUBJECT) is
      -- Update subscribed observers because a subject's state changed.
    deferred
    end
end -- class OBSERVER

class
  CONCRETE_OBSERVER
inherit
  OBSERVER
create
  make

feature {NONE} -- Initialization
  make is
      -- Create subject_1 and subject_2.
    do
      create subject_1.make (Current)
      create subject_2.make (Current)
    end

feature -- Access
  subject_1: SUBJECT_1
      -- First subject of observer
  subject_2: SUBJECT_2
      -- Second subject of observer

feature -- Basic operations
```

```
update (a_subject: SUBJECT) is
      -- Update subscribed observers because a subject's state changed.
   do
 ...
   end

end -- class CONCRETE_OBSERVER

deferred class
   SUBJECT

feature {NONE} -- Initialization
   make (an_observer: like observer) is
      -- Set observer to an_observer.
   require
      an_observer_not_void: an_observer /= Void
   do
      observer := an_observer
   ensure
      observer_set: observer = an_observer
   end

feature -- Access
   observer: OBSERVER
   -- OBSERVER

feature -- Mediator pattern
   notify is
      -- Notify observer that current subject has changed.
   do
      observer.update (Current)
   end

   do_something is
     -- Do something.
   deferred
   end

invariant
   observer_not_void: observer /= Void

end -- class SUBJECT

class
   SUBJECT_1
inherit
   SUBJECT

create
   make

feature - Basic elements
   do_something is
      -- Do something.
   do
      io.put_string ("This is the first subject")
      io.new_line
   end
   change is
      -- Change the state of the object
   do
      -- ...
```

```
            notify
        end

end -- class SUBJECT_1


class
    SUBJECT_2
inherit
    SUBJECT

create
    make

feature - Basic elements
    do_something is
            -- Do something.
        do
            io.put_string ("This is the second subject")
            io.new_line
        end
    change is
            -- Change the state of the object
        do
            -- ...
            notify
        end

end -- class SUBJECT_2
```

The Observer design pattern uses a notify-update mechanism. Replace this notify-update mechanism by using the *EVENT_TYPE* class for the above application. The interface of class *EVENT_TYPE* is given below:

```
class interface
    EVENT_TYPE [EVENT_DATA -> TUPLE create default_create end]

feature -- Element change

    subscribe (an_action: PROCEDURE [ANY, EVENT_DATA]) is
            -- Add an_action to the subscription list.
        require
            an_action_not_void: an_action /= Void
            an_action_not_already_subscribed: not has (an_action)
        ensure
            an_action_subscribed: count = old count + 1 and has (an_action)
            index_at_same_position: index = old index

    unsubscribe (an_action: PROCEDURE [ANY, EVENT_DATA]) is
            -- Remove an_action from the subscription list.
        require
            an_action_not_void: an_action /= Void
            an_action_already_subscribed: has (an_action)
        ensure
            an_action_unsubscribed: count = old count - 1 and not has (an_action)
            index_at_same_position: index = old index
        end

feature -- Publication

    publish (arguments: EVENT_DATA) is
            -- Publish all actions from the subscription list .
        require
```

```
        arguments_not_void: arguments /= Void

feature -- Measurement

    count: INTEGER
        -- Number of items

    index: INTEGER is
        -- Index of current position in the list of actions

feature -- Access

    has (v: PROCEDURE [ANY, EVENT_DATA]): BOOLEAN
        -- Does the list of actions include v?

end -- class EVENT_TYPE
```

. . . .

. . . .

. . . .

. . . .

. . . .

5 Concurrent Programming (17 Points)

5.1 True or false (3 Points)

Are the following statements true or false? Write “T” for true or “F” for false in the corresponding box.

Answer	*Statement*
	The exact execution path of a concurrent program is non-deterministic in general, even with the same input.
	Access to and modification of shared variables should always be mutually exclusive.
	The sequence of instructions protected by a semaphore (through a “wait” operation) can be executed by at most one thread of control at any time.

5.2 Busy waiting (3 points)

Explain what busy waiting is and how semaphores remove the need for busy waiting.

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

5.3 Semaphores (3 points)

A semaphore has an associated integer variable. Explain under what conditions the value of that variable can be: 1) Positive 2) Zero.

. . . .

. . . .

. . . .

. . . .

. . . .

5.4 Programming (8 points)

Consider the following scenario. There is a printing server which puts print tasks into a buffer, and a printer which gets printing tasks from the buffer, one at a time. If there is no task in the buffer, the printer will wait. The buffer is assumed to have infinite length and should be accessed exclusively.

Complete the following program using semaphore(s) or mutex(es) to make sure the printing server and the printer can cooperate correctly. You can assume that if S is a semaphore or a mutex, the calls *wait* (*S*) and *signal* (*S*) are available with the usual semantics.

Semaphore(s) or mutex(es) definition:

. . . .

. . . .

Printing server program:

new_task := *next_print_task* −− Get a new print task.

. . . .

. . . .

store_task (*new_task*, *buffer*) −− Store the print task into buffer.

. . . .

. . . .

Printer program:

....

....

print_task := *task_from_buffer* (*buffer*) −− Get a print task from buffer.

remove_task (*print_task*, *buffer*) −− Remove the print task from buffer.

....

....

print (*print_task*) −− Process the task.