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!
<table>
<thead>
<tr>
<th>Question</th>
<th>Number of possible points</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>
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 40th 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 × STRING × INTEGER × INTEGER → CD
- title: CD → STRING
- artist: CD → STRING
- price: CD → INTEGER
- quantity: CD → INTEGER
- set_price: CD × INTEGER → CD
- sell: CD × INTEGER → CD
- order_new: CD × INTEGER → 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
2    description: "Objects that represent a natural counter."

4 class COUNTER
6 create make
8 feature -- Initialization
10      make is
12          -- Create a counter initializing it with 0.
14          do
15            item := 0
16          ensure
17            initial_value_is_0 : item = 0
18      end
18 feature -- Element change
20      increment is
22          -- Increment the counter by 1.
24          do
25            item := item + 1
26          ensure
28      end
28 decrement is
30          -- Decrement the counter by 1.
32          require
34          do
35            item := item - 1
36          ensure
38      end
40 feature -- Implementation
42      item: INTEGER
44 invariant
46
48 end

indexing
2    description: "Objects that represent students. A student consists of a name."

4 class STUDENT
6 create make
8 feature -- Initialization
10      make (n: STRING) is
class COURSE

create

make {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

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"

create

make


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

  end

  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

  class interface
CREATE LINKED_LIST [G]

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?
    istlast: 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.

forth — Move cursor to next position.

go_to (p: CURSOR)

start — Move cursor to first position.

search (v: like item) is

merge_left (other: like Current)

merge_right (other: like Current)

put_front (v: like item)

replace (v: like item)

feature — Element change

extend (v: like item)

merge_left (other: like Current)

merge_right (other: like Current)

put_front (v: like item)

replace (v: like item)

feature — Removal

remove

prune (v: like item) is

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

3. Structural design patterns:
   - Name: ....................................................... 
     Description: ............................................
4 Observer (18 Points)

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

```plaintext
defered class
  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
    end
  feature -- Access
    subject_1: SUBJECT_1
      -- First subject of observer
    subject_2: SUBJECT_2
      -- Second subject of observer
  feature -- Basic operations
```

---

*ETHZ D-INFK*

*Prof. Dr. B. Meyer*  
Software Architecture – Exam
update ( a_subject: SUBJECT) is
   -- Update subscribed observers because a subject's state changed.
   do
      ...
   end
end -- class CONCRETE_OBSERVER

defered 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_not: 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
      -- ...

The Observer design pattern uses a notify-update mechanism. Replace this notify-update mechanism by using the \texttt{EVENT\_TYPE} class for the above application. The interface of class \texttt{EVENT\_TYPE} is given below:

\begin{verbatim}
class interface
EVENT\_TYPE [EVENT\_DATA -> TUPLE create default create end]
end

feature -- Publication
publish (arguments: EVENT\_DATA) is
  -- Publish all actions from the subscription list.
  require
end

feature -- 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
end

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 -- 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_1
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
\end{verbatim}
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.

<table>
<thead>
<tr>
<th>Answer</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The exact execution path of a concurrent program is non-deterministic in general, even with the same input.</td>
</tr>
<tr>
<td></td>
<td>Access to and modification of shared variables should always be mutually exclusive.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

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:**

\[
\text{print}_\text{task} := \text{task}_\text{from_buffer} (\text{buffer}) \quad -- \quad \text{Get a print task from buffer.}
\]

\[
\text{remove}_\text{task} (\text{print}_\text{task}, \text{buffer}) \quad -- \quad \text{Remove the print task from buffer.}
\]

\[
\text{print} (\text{print}_\text{task}) \quad -- \quad \text{Process the task.}
\]