Software Architecture Exam

Spring Semester 2009
Prof. Dr. Bertrand Meyer
Date: 26 May 2009

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: 120 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>18</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
1 Multiple choice questions (18 points)

For each statement found below, indicate through a checkmark in the corresponding column whether it is false or true. For each statement, you can mark at most one square. A correctly set checkmark is worth 1 point, an incorrectly set checkmark is worth 0 points.

Example:

Which of the following statements are true and which are false for objects and classes of Eiffel?

<table>
<thead>
<tr>
<th>True</th>
<th>False</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑</td>
<td>☐</td>
<td>a. Classes exist only in the software text; objects exist only during the execution of the software.</td>
</tr>
<tr>
<td>☐</td>
<td>☑</td>
<td>b. Each object is an instance of its generic class.</td>
</tr>
<tr>
<td>☐</td>
<td>☑</td>
<td>c. An object is deferred if it has at least one deferred feature.</td>
</tr>
</tbody>
</table>

1.1 Which of the following statements are true and which are false for the Unified Modeling Language?

<table>
<thead>
<tr>
<th>True</th>
<th>False</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑</td>
<td>☐</td>
<td>a. A UML model is a set of classes and relations between them.</td>
</tr>
<tr>
<td>☑</td>
<td>☐</td>
<td>b. The general semantics of the dependency relation is that if the independent entity changed, the dependent one may also change.</td>
</tr>
<tr>
<td>☐</td>
<td>☑</td>
<td>c. State diagrams are intended for describing dynamic behavior of the system.</td>
</tr>
<tr>
<td>☐</td>
<td>☑</td>
<td>d. An association between two classes on a class diagram means that they are different implementations of the same abstraction.</td>
</tr>
<tr>
<td>☑</td>
<td>☐</td>
<td>e. Aggregation expresses the ”part-of” relation, which means that the part is always created and destroyed together with the aggregate.</td>
</tr>
<tr>
<td>☑</td>
<td>☐</td>
<td>f. In contrast to communication diagrams, on sequence diagrams not only the set of entities and relations between them matters, but also the spatial placement of elements.</td>
</tr>
</tbody>
</table>

1.2 Which of the following statements are true and which are false for the Eiffel exception mechanism?

<table>
<thead>
<tr>
<th>True</th>
<th>False</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑</td>
<td>☐</td>
<td>a. The execution of a rescue clause must in all cases re-establish the class invariant.</td>
</tr>
<tr>
<td>☑</td>
<td>☐</td>
<td>b. If a retry succeeds, the program execution continues normally.</td>
</tr>
<tr>
<td>☑</td>
<td>☐</td>
<td>c. If a rescue clause only contains a retry, then the retry will be executed at most once.</td>
</tr>
<tr>
<td>☑</td>
<td>☐</td>
<td>d. If a rescue clause only contains a retry, then the retry will be repeatedly executed until there is no failure any more.</td>
</tr>
<tr>
<td>☑</td>
<td>☐</td>
<td>e. If an exception is triggered in a routine that doesn’t have a rescue clause, then the exception is passed to the caller.</td>
</tr>
</tbody>
</table>
1.3 If a software element C (client) needs a service from a software element S (supplier), the following four possibilities exist:

1. C must know the identity of S and S must know the identity of C.
2. C must know the identity of S, but S does not have to know the identity of C.
3. S must know the identity of C, but C does not have to know the identity of S.
4. Neither needs to know the identity of the other.

State which one of 1, 2, 3 or 4 applies to the following architecture styles:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>a. Batch-sequential</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>b. Pipe-and-filter</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>c. Call-and-return</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>d. Event-based (Publish-Subscribe)</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>e. Blackboard</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>f. Hierarchically layered</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>g. Client-Server</td>
</tr>
</tbody>
</table>

2 Abstract Data Types and Design by Contract (20 Points)

2.1 Incompleteness in contracts (3 Points)

Tic-Tac-Toe game is played on a 3-by-3 board, which is initially empty. There are two players: a “cross” player and a “circle” player. They take turns; each turn changes exactly one cell on the board from empty to the symbol of the current player (cross or circle). The “cross” player always starts the game. The rules that define when the game ends and which player wins are omitted from the task for simplicity.

Below you will find an interface view of GAME class representing Tic-Tac-Toe games.

class GAME

create make

feature -- Initialization
make
  -- Create an empty 3-by-3 board
  ensure
cross_turn: next_turn = Cross
end

feature -- Constants
Empty: INTEGER is 0
Cross: INTEGER is 1
Circle: INTEGER is 2
  -- Symbolic constants for players and states of board cells

feature -- Access
next_turn: INTEGER
   -- Player that will do the next turn

item (i, j: INTEGER): INTEGER
   -- Value in the board cell (i, j)
   require
      i_in_bounds: i >= 1 and i <= 3
      j_in_bounds: j >= 1 and j <= 3
   ensure
      valid_value: Result = Empty or Result = Cross or Result = Circle
end

feature — Basic operations
put_cross (i, j: INTEGER)
   -- Put cross into the cell (i, j)
   require
      cross_turn: next_turn = Cross
      i_in_bounds: i >= 1 and i <= 3
      j_in_bounds: j >= 1 and j <= 3
      empty: item (i, j) = Empty
   ensure
      cross_put: item (i, j) = Cross
      circle_turn: next_turn = Circle
end

put_circle (i, j: INTEGER)
   -- Put circle into the cell (i, j)
   require
      circle_turn: next_turn = Circle
      i_in_bounds: i >= 1 and i <= 3
      j_in_bounds: j >= 1 and j <= 3
      empty: item (i, j) = Empty
   ensure
      circle_put: item (i, j) = Circle
      cross_turn: next_turn = Cross
end

invariant
   valid_player: next_turn = Cross or next_turn = Circle
end

The contract of this class is incomplete with respect to the game description
given above. In which contract elements does the incompleteness reside? Ex-
press in natural language what the missing parts of the specification are. Give
an example of a scenario that is allowed by the above contract, but should not
happen in Tic-Tac-Toe:
2.2 ADT GAME (10 Points)

Create an ADT that describes Tic-Tac-Toe games. The ADT functions should correspond one-to-one to the features of the GAME class above. The axioms of the ADT should be sufficiently complete, overcoming the incompleteness of the class contracts.

TYPES
GAME

FUNCTIONS
- make :
- next_turn :
- item :
- put_cross :
- put_circle :
- Empty :
- Cross :
- Circle :

PRECONDITIONS
P1
P2
P3

AXIOMS
A1
A2
A3
A4
A5
2.3 Proof of sufficient completeness (7 Points)

Prove that your specification is sufficiently complete.
3 Design patterns I (22 Points)

Given is a class hierarchy that models a very simple forum system. It consists of three classes: class \texttt{FORUM_ENTITY}, class \texttt{POST} and class \texttt{THREAD} (see Listings 1, 2, and 3).

\textbf{Question 1:} The classes \texttt{FORUM_ENTITY}, \texttt{POST} and \texttt{THREAD} have been prepared for the implementation of a Visitor pattern, but they also implement a second pattern from the lecture. Which design pattern? Give its name.

\texttt{Listing 1: Class FORUM_ENTITY}

\begin{verbatim}
def class FORUM_ENTITY
  feature -- Access
  title : STRING
    -- Title of entity
  owner: STRING
    -- Username of person that initiated the thread/post
  output: STRING
    -- Textual description
  deferred
end

feature -- Status report
is_private: BOOLEAN
  -- Is entity read restricted?

feature -- Element setting
set_title_and_owner (t, o: STRING)
  -- Set 'title' to 't' and 'owner' to 'o'.
require
  \_valid : t /= Void and then not t.is_empty
  o_valid: o /= Void and then not o.is_empty
do
  title := t
  owner := o
ensure
  title_set : t.is_equal(title)
  owner_set: o.is_equal(owner)
end
\end{verbatim}
set_private (b: BOOLEAN)
   -- Set 'is_private'.
   do
      is_private := b
   ensure
      is_private_set : is_private = b
   end

feature -- Basic operations

process (v: VISITOR)
   -- Process 'Current' with visitor 'v'.
   require
      v_exists : v /= Void
   deferred
   end
end

Listing 2: Class POST

class POST inherit FORUM_ENTITY

create
set_title_and_owner

feature -- Access

text: STRING
   -- Message of post

feature -- Element change

set_text (s: STRING)
   -- Set 'text' to 's'.
   require
      s_valid: s /= Void and then not s.is_empty
   do
      text := s
   ensure
      text_set: s.is_equal (text)
   end

feature -- Basic operations

output: STRING
   -- Textual description
   do
      Result := "********** POST **********%NTitle: " + title + 
               "%ONOwner: " + owner + "%N"
      if text /= Void then
         Result := Result + text + "%NN"
      end
   end

process (v: VISITOR)
   -- Process 'Current' with visitor 'v'.
   do
   end
end
Listing 3: Class THREAD

```plaintext
class THREAD inherit
  FORUM_ENTITY
  redefine
    set_title_and_owner
  end

create
  set_title_and_owner

feature -- Access
  contents: ARRAYED_LIST[FORUM_ENTITY]
  -- Contents of the thread

feature -- Element change
  set_title_and_owner (t, o: STRING)
  -- Set 'title' to 't' and 'owner' to 'o'.
  do
    Precursor (t, o)
    create contents.make (5)
  end

add_entity (e: FORUM_ENTITY)
  -- Add 'e' to last position of 'contents'.
  require
    not_there: not contents.has (e)
  do
    contents.force (e)
  end

feature -- Basic operations
output: STRING
  -- Textual description
  do
    Result := "********** THREAD **********\nTitle: " + title + \nOwner: " + owner + "\n
process (v: VISITOR)
  -- Process 'Current' with visitor 'v'.
  do
    .................................................................
  end
end
```

Question 2: Complete the implementation of the visitor pattern by filling in the missing lines in the classes POST and THREAD and by providing the code of VISITOR and READ_VISITOR. The main goal of READ_VISITOR is the generation of output for a hierarchy of threads and posts. It should show the following characteristics:

- The call entity.process (v) with entity of type FORUM_ENTITY and v of type READ_VISITOR should do a depth first traversal of the hierarchy attached to entity.
• During the traversal, it calls \texttt{output} on a visited entity if either (a) the entity is not private (see feature \texttt{is\_private} of class \texttt{FORUM\_ENTITY}) or (b) the \texttt{READ\_VISITOR} has access to private entities (see feature \texttt{has\_private\_access} of class \texttt{READ\_VISITOR}). The output is collected in the variable \texttt{last\_output} of \texttt{READ\_VISITOR}. 
Listing 4: Class \texttt{VISITOR}

\begin{verbatim}
deferred class VISITOR
end
\end{verbatim}
Listing 5: Class READ_VISITOR

class READ_VISITOR inherit VISITOR

create

double feature -- Initialization

make (b: BOOLEAN)
  -- Initialize and set flag for reading private threads and posts.
  do
    last_output := ""
  end
  has_private_access := b
  ensure
  output_exists: last_output /= Void
  private_access_set: has_private_access = b
end

feature -- Access

last_output: STRING

double feature -- Status report

has_private_access: BOOLEAN

double feature (FORUM_ENTITY) -- Basic operations
Question 3: Listing 6 shows the root class \textit{APPLICATION} of a system that provides a user interface to log in and out of the system and print the hierarchy of threads and posts. The feature \textit{prepare} reads a hierarchy of threads and posts from a file (contents are omitted). Redesign the class to use a pattern that helps removing the case distinctions between a logged in user and an anonymous user found in the features \textit{login}, \textit{logout} and \textit{read_entity}.

What pattern would you use? Give its name. .......................... 

Draw a diagram of the involved classes and list the names of all their features. A partial version of \textit{APPLICATION} is given as a starting point. Explain in a couple of sentences how the involved classes interact and why the case distinctions disappear.

\begin{center}
\begin{tikzpicture}
\node [rectangle, draw] (A) {\textit{APPLICATION}};
\node [rectangle, draw, below of=A] (B) {prepare\textit{}};
\node [rectangle, draw, below of=B] (C) {entity: \textit{ENTITY}};
\end{tikzpicture}
\end{center}
Listing 6: Class APPLICATION

class APPLICATION

create
make

feature -- Access

is_logged_in : BOOLEAN
--- Is user logged in?

entity: FORUM_ENTITY
--- Top level entity

username: STRING
--- Username of logged in user (may be void)

feature -- Basic operations

login
--- Log in if not already logged in.
do
if not is_logged_in then
io.puts_string ("Username: ")
io.read_word
if not io.last_string .is_empty then
username := io.last_string
is_logged_in := True
else
io.puts_string ("Username invalid")
end
else
io.puts_string ("You have to logout first.")
end
ensure
username_set: username /= Void
end

logout
--- Log out if logged in.
do
if not is_logged_in then
io.puts_string ("You have to login first.")
else
username := Void
is_logged_in := False
end
ensure
username_set: username = Void
end

read_entity
--- Read entity contents.
local
v: READ_VISITOR
do
if is_logged_in then
create v.make (True)
else
create v.make (False)
end
entity.process (v)
62     io. put_string (v. last_output) 
63     end
64 
65     feature -- Initialization
66     make is
67     -- Run application.
68     local 
69     c: CHARACTER 
70     do
71     prepare 
72     from
73     io. put_string ( "%N>" )
74     io. read_character
75     c := io. last_character
76     until 
77     c = 'q'
78     loop
79     inspect c 
80     when 'i' then 
81     login 
82     when 'o' then 
83     logout 
84     when 'r' then 
85     read_entity 
86     else
87     if c. is_alpha then
88     io. put_string ( "Available commands: %Ni: login%No: logout%Nr: 
read entity%Nq: quit%N"")
89     end
90     end
91     if c. is_alpha then
92     io. put_string ( "%N>")
93     end
94     io. read_character
95     c := io. last_character
96     end
97     end
98     end
99     end
100    feature {NONE} -- Implementation 
101     prepare
102     -- Fill some threads and posts. 
103     local
104     top, sub1, sub2: THREAD 
105     p: POST 
106     do
107     -- Implementation removed to improve readability. 
108     ensure
109     entity_exists : entity /= Void
110     end
111     end
112     end

4 Design Patterns II (16 Points)

A company selling furniture has an interactive program to show customers what a furnished room would look like. Furniture pieces can be added and removed from the room, and these actions can be undone and redone. Here is a typical
The room uses an abstract factory (an instance of FURNITURE_FACTORY) to create furniture pieces (instances of CHAIR and DESK). Assume the following classes:

defered class
1  FURNITURE_FACTORY

defered class
2  FURNITURE PIECE
Your task is to implement the command pattern that supports the undo-redo mechanism by filling in code in classes ROOM, ADD_ACTION and REMOVE_ACTION. Since these classes cooperate closely, it’s a good idea to study them carefully before writing the code. Some features of classes LINKED_LIST and STACK that you might find useful are shown at the end.

```plaintext
defered class ACTION
    feature
        perform deferred end
        unperform deferred end
end

class ADD_ACTION
    inherit ACTION
    create make

    feature
        make (n: INTEGER; fp: FURNITURE_PIECE; l: LIST [TUPLE [id: INTEGER; f: FURNITURE_PIECE]])
        -- Initialize an add-action into ‘l’ of ‘fp’ with ID ‘n’.
        require
        do
            piece_number := n
            furniture_piece := fp
            furniture_list := l
        end
```
perform
  -- Add the furniture piece to the room.

unperform
  -- Undo the last ‘perform’.
  do
    furniture_list . prune_all ([piece_number, furniture_piece])
  end

feature {NONE} -- Implementation
  piece_number: INTEGER
  furniture_piece: FURNITURE_PIECE
  furniture_list : LIST [TUPLE [id: INTEGER; fp: FURNITURE_PIECE]]
  -- The room’s contents.

invariant
  furniture_list_exists : furniture_list /= Void

end

class
  REMOVE_ACTION
    inherit ACTION
    create
      make
        feature (n: INTEGER; l: LIST [TUPLE [id: INTEGER; fp: FURNITURE_PIECE]])
          -- Initialize an action to remove a furniture piece with ID ‘n’ from ‘l’.
          require
            furniture_list_exists : l /= Void
          do
            piece_number := n
            furniture_list := l
          end
        perform
          -- Remove the furniture piece from the room, if possible.
          local
            found: BOOLEAN
            furniture_item: TUPLE [n: INTEGER; fp: FURNITURE_PIECE]
          do
            from
              furniture_list . start
              furniture_piece := Void
            until
              found or else furniture_list . off
            loop
furniture_item := furniture_list . item
if furniture_item . n = piece_number then
    furniture_piece := furniture_item . fp
    furniture_list . remove
    found := True
end
if not found then
    furniture_list . forth
end

unperform -- Undo the last 'perform'.
do

feature {NONE} -- Implementation
piece_number: INTEGER
furniture_piece: FURNITURE_PIECE
furniture_list: LIST [TUPLE [id: INTEGER; f: FURNITURE_PIECE]]
-- The room's contents.

invariant
furniture_list_exists : furniture_list /= Void

class
ROOM

create
make_with_furniture_factory

feature
make_with_furniture_factory (f: FURNITURE_FACTORY)
-- Create an empty room.
require
factory_exists : f /= Void
do
furniture_factory := f
create {LINKED_LIST [TUPLE [INTEGER, FURNITURE_PIECE]]}
furniture_list.make
furniture_list . compare_objects
-- Use object rather than reference comparison for elements.
id_counter := 1
create {LINKED_STACK [ACTION]} undoable_action_stack.make
create {LINKED_STACK [ACTION]} redoable_action_stack.make

ensure

    room_empty: furniture_piece_count = 0

end

add_chair (c: COORDINATE)
    -- Add a chair to the room at coordinate ‘c’.
    local
        add_action: ADD_ACTION
        chair: CHAIR
        do
            furniture_factory.make_chair (c)
            chair := furniture_factory.made_chair

    added_chair_id := id_counter
    id_counter := id_counter + 1

ensure

    added_chair: furniture_piece_count = old furniture_piece_count + 1

end

add_desk (c: COORDINATE)
    -- Add a desk to the room.
    local
        add_action: ADD_ACTION
        desk: DESK
        do
        -- Implementation not shown.
        ensure

            added_desk: furniture_piece_count = old furniture_piece_count + 1

end

remove_chair (n: INTEGER)
    -- Remove chair with id ‘n’ from the room.
    -- Do nothing if the chair is not inside the room.
    local
        remove_action: REMOVE_ACTION
        do

        ensured

end
68

---

70

---

72

---

74

\begin{verbatim}
ensure
possibly_removed_chair: furniture_piece_count <= old furniture_piece_count
end
\end{verbatim}

76

\begin{verbatim}
remove_desk (n: INTEGER)
  -- Remove desk with id 'n' from the room.
  -- Do nothing if the desk is not inside the room.
local
  remove_action: REMOVE_ACTION
  do
    create remove_action.make (n, furniture_list)
    remove_action.perform
    undoable_action_stack.put (remove_action)
    redoable_action_stack.wipe_out
ensure
possibly_removed_desk: furniture_piece_count <= old furniture_piece_count
end
\end{verbatim}

78

\begin{verbatim}
undo
  -- Undo the last add or remove action.
local
  action: ACTION
  do
\end{verbatim}

80

---

82

---

84

---

86

---

88

---

90

---

92

---

94

---

96

---

98

---

100

---

102

---

104

---

106

---

108

---

110

\begin{verbatim}
end
\end{verbatim}
redo
  -- Redo the last undone action.
local
  action: ACTION
do
  if not redoable_action_stack.is_empty then
    action := redoable_action_stack.item
    redoable_action_stack.remove
    action.perform
    undoable_action_stack.put (action)
end
end

added_chair_id: INTEGER
  -- A handle for the last added chair.
added_desk_id: INTEGER
  -- A handle for the last added desk.

furniture_piece_count : INTEGER
  -- The number of furniture pieces inside.
do
  Result := furniture_list.count
end

feature { None } -- Implementation
  furniture_factory: FURNITURE_FACTORY
  furniture_list: LIST [TUPLE [INTEGER, FURNITURE PIECE]]
  id_counter: INTEGER
    -- Internal counter to provide handles to created furniture pieces.
undoable_action_stack: STACK [ACTION]
  -- Stack storing done actions that can be undone.
redoable_action_stack: STACK [ACTION]
  -- Stack storing undone actions that can be redone.

invariant
  furniture_factory_exists : furniture_factory /= Void
  furniture_list_exists : furniture_list /= Void
  undoable_action_stack_exists : undoable_action_stack /= Void
  redoable_action_stack_exists : redoable_action_stack /= Void

end

1 class
  LINKED_LIST [G]

  feature -- General operations.
  force (v: G)
    -- Add 'v' to end.
    require
      extendible: extendible
    ensure
      new_count: count = old count + 1
      item_inserted: has (v)
  prune_all (v: G)
    -- Remove all occurrences of 'v'.

  feature -- Cursor-based operations.
class STACK [G]

feature

put (v: G)
  -- Push 'v' onto top.
  ensure
  item.pushed: stem = v

remove
  -- Remove the top item.

wipe_out
  -- Remove all items.
  ensure
  wiped_out: is.empty

item: G
  -- The top element of the stack.

is_empty: BOOLEAN
  -- Is the stack empty?

-- Other features omitted.