Software Architecture Exam

Summer Semester 2006
Prof. Dr. Bertrand Meyer
Date: 5 July 2006

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>9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>
1 Design by Contract, software lifecycle model, configuration management (9 points)

Put checkmarks in the checkboxes corresponding to the correct answers. Multiple correct answers are possible; there is at least one correct answer per question. A correctly set checkmark is worth 1 point, an incorrectly set checkmark is worth -1 point. If the sum of your points is negative, you will receive 0 points.

Example:

1. Which of the following statements are true?
   a. Classes exist only in the software text; objects exist only during the execution of the software.  ✓
   b. Each object is an instance of its generic class.  □
   c. An object is deferred if it has at least one deferred feature.  □

1. Design by Contract. The class invariant must be satisfied...
   a. after any qualified call to any feature of the class.  □
   b. after any call to any feature of a class.  □
   c. after object creation.  □
   d. only after calls to features exported to ANY.  □

2. Lifecycle models. Which of the following statements are true?
   a. The waterfall model is synchronous and has the disadvantage that the actual code appears late in the development process.  □
   b. The cluster model adds a generalization task to the waterfall model. Therefore the steps in the cluster model can be parallelized.  □
   c. Lifecycle models aim at improving the quality of the software system in general and the process of software development in particular.  □
   d. When the lifecycle of a software system is over, it transits to a new lifecycle model.  □

3. Configuration management. Version numbers...
   a. must be part of the file name.  □
   b. form a partial ordering.  □
   c. are always INTEGER numbers, starting with 1.  □
   d. are linked to a point in time.  □

4. Configuration management. Which parts of the development process are part of configuration management:
   a. UML design diagrams  □
   b. project budget  □
   c. daily/nightly build results  □
   d. bug reports by users  □
2 Modularity and reusability (9 points)

2.1 Correctness vs. robustness (4 points)

Define Software Correctness:

Define Software Robustness:

Give an example illustrating the difference between Software Robustness and Correctness:

2.2 Modularity principles I (2 points)

The inheritance mechanism of Eiffel implements one of the modularity principles. Mention which one and explain how inheritance is used in this principle.
2.3 Modularity principles II (3 points)

For the code snippet below explain which modularity principle it violates. Explain the principle and then show how to correct the code snippet.

```育
class DATABASE

feature -- Element change

store (key: INTEGER; value: ANY) is
    do ...
end

feature -- Access

select (key: INTEGER): ANY is
    require
        key_valid: table.has(key)
    do ...
end

table: HASH_TABLE [ANY, INTEGER]
    -- Data storage

end
```

Modularity principle that is violated:

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

Explanation of the modularity principle:
3 Abstract Data Types (22 Points)

The following abstract data type models a file system. The file system stores data under filenames. Write operations either create files or overwrite existing ones. It is not possible to change only parts of an existing file. The file system does not offer directories. Types NAME and DATA are assumed to be defined separately; their actual content is not visible and does not matter for the exercise.

**TYPES**

FILE_SYSTEM, DATA, NAME

**FUNCTIONS (all provisionally marked total)**

format_disk: FILE_SYSTEM
write: NAME × DATA × FILE_SYSTEM → FILE_SYSTEM
read: NAME × FILE_SYSTEM → DATA
file_exists: NAME × FILE_SYSTEM → BOOLEAN

**PRECONDITIONS** (n ∈ NAME; f ∈ FILE_SYSTEM)
read \((n, f)\) require file\_exists\((n, f)\)

**AXIOMS** \((d \in \text{DATA}; n, m \in \text{NAME} \text{ with } n \neq m; f \in \text{FILE\_SYSTEM})\)

- \(\text{file\_exists} \ (n, \text{format\_disk}) = \text{false}\) (Axiom 1)
- \(\text{file\_exists} \ (n, \text{write} \ (n, d, f)) = \text{true}\) (Axiom 2)
- \(\text{file\_exists} \ (n, \text{write} \ (m, d, f)) = \text{file\_exists} \ (n, f)\) (Axiom 3)
- \(\text{read} \ (n, \text{write} \ (n, d, f)) = d\) (Axiom 4)
- \(\text{read} \ (n, \text{write} \ (m, d, f)) = \text{read} \ (n, f)\) (Axiom 5)

**To Do:**

1. In the **FUNCTIONS** paragraph above all functions are shown as total, but some should be partial. Mark those which should be partial (by crossing the corresponding arrow) (2 points).

2. In the following equations, \(d_1,d_2 \in \text{DATA} \text{ with } d_1 \neq d_2; n, m \in \text{NAME} \text{ with } n \neq m; f \in \text{FILE\_SYSTEM}\). For each of the equations, prove one of the following: (1) the equation is not correct; (2) it is correct and does not hold; (3) it is correct and holds (4 points).
   (a) \(\text{read} \ (n, \text{write} \ (n, d_1, f)) = d_1\)
   
   ____________________________________________
   ____________________________________________
   ____________________________________________
   ____________________________________________
   ____________________________________________
   ____________________________________________
   ____________________________________________
   ____________________________________________

   (b) \(\text{read} \ (n, \text{write} \ (m,d_2,\text{write} \ (n,d_1,f))) = \text{read} \ (n,\text{write} \ (n,d_1,\text{write} \ (m,d_2,f)))\)
   
   ____________________________________________
   ____________________________________________
   ____________________________________________
   ____________________________________________
   ____________________________________________
   ____________________________________________
   ____________________________________________
   ____________________________________________
(c) \( \text{read} \ (n, \text{write} \ (n, \text{d2}, \text{write} \ (n, \text{d1}, f))) = \text{read} \ (n, \text{write} \ (n, \text{d1}, \text{write} \ (n, \text{d2}, f)) \)

(d) \( \text{read} \ (n, \text{write} \ (m, \text{d1}, \text{formatDisk})) = \text{d1} \)
3. Prove the sufficient completeness of the abstract data type (8 points).
4. We want to extend the model with a function describing the deletion of an existing file. A deleted file does not exist anymore and is thus not readable.

(a) Add a new function that models this operation.
(b) Adapt the existing preconditions accordingly.
(c) Define the necessary axioms.
(d) Give an argument (a complete proof is not required) explaining why your extension still satisfies sufficient completeness.

(8 points)
4 Design Patterns (18 Points)

The Mediator pattern “defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently” (“Design Patterns. Elements of reusable Object-Oriented Software”, E. Gamma et al., Addison-Wesley, 1995).

The Mediator pattern describes a way to control the interactions between a set of objects called “colleagues”. Rather than having everyone know everyone else, a central point of contract (the “mediator”) knows about its “colleagues”.

In a system designed according to the Mediator pattern, colleagues only know about their mediator: they send requests to the mediator, which takes care of forwarding them to the appropriate colleague; the requested colleague also sends its answer back to the mediator, which forwards it to the originator of the request. There is no direct interaction between colleagues. Everything goes through the mediator.

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

1 deferred class
MEDIATOR
3 feature -- Basic operations
    update_colleagues (a_colleague: COLLEAGUE) is
5    -- Update colleagues because a_colleague changed.
        deferred
class MY_MEDIATOR
  inherit MEDIATOR
  create
    make
  feature {NONE} -- Initialization
    make is
      -- Create colleague_1 and colleague_2.
      do
        create colleague_1.make (Current)
        create colleague_2.make (Current)
      end
  feature -- Access
    colleague_1: COLLEAGUE_1
      -- First colleague of mediator
    colleague_2: COLLEAGUE_2
      -- Second colleague of mediator
  feature -- Basic operations
    update_colleagues (a_colleague: COLLEAGUE) is
      -- Update colleagues because a_colleague changed.
      do
        if a_colleague = colleague_1 then
          colleague_2.do_something
        elseif a_colleague = colleague_2 then
          colleague_1.do_something
        end
      end
end -- class MY_MEDIATOR

defered class COLLEAGUE
  feature {NONE} -- Initialization
    make (a_mediator: like mediator) is
/* Set mediator to \texttt{a\_mediator}.

\begin{verbatim}
require
  a\_mediator\_not\_void: a\_mediator /= Void
do
  mediator := a\_mediator
ensure
  mediator\_set: mediator = a\_mediator
end
\end{verbatim}

feature -- Access
mediator: MEDIATOR
  -- Mediator

feature -- Mediator pattern

\begin{verbatim}
notify\_mediator is
  -- Notify mediator that current colleague has changed.
do
  mediator.update\_colleagues (Current)
end
\end{verbatim}

do\_something is
  -- Do something.
defered
end

\begin{verbatim}
invariant
  mediator\_not\_void: mediator /= Void
end
-- class COLLEAGUE
\end{verbatim}

class COLLEAGUE
  inherit COLLEAGUE
create
make

feature -- Basic elements
do\_something is
  -- Do something.
\begin{verbatim}
do
  io.put\_string ("This is colleague 1")
io.new\_line
end
\end{verbatim}

change is
  -- Change the state of the object
do
  -- ...
end
-- class COLLEAGUE\_2

\begin{verbatim}
class COLLEAGUE\_2
  inherit COLLEAGUE
create
make
\end{verbatim}
feature -- Basic elements
   do_something is
   -- Do something.
   do
      io.put_string("This is colleague 2")
      io.new_line
   end
change is
   -- Change the state of the object
   do
      notify_mediator
   end
end -- class COLLEAGUE 2

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

class
   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 not suspended 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 Design by Contract (10 Points)

A binary tree is a tree data structure in which each node has at most two children. Typically the child nodes are called left and right. In the following class implementing this notion, complete the contracts at the locations marked by dotted lines.

```java
class BINARY_TREE[G]
inherit CELL[G]
    undefined
    copy, is_equal
end
TREE[G]
redefine
    parent,
    is_leaf,
    subtree_has,
    subtree_count,
    fill_list,
    child_remove,
    child_after,
    child_capacity,
    tree_copy,
    child_start,
    child_forth
```
end

create

make

feature -- Initialization

make (v: like item) is

do
item := v

ensure

end

feature -- Access

parent: BINARY_TREE [G]

-- Parent of current node

titem: G

-- Item in current node

child_index: INTEGER

-- Index of cursor position

left_child: like parent

-- Left child, if any

right_child: like parent

-- Right child, if any

left_item: like item is

-- Value of left child

require

result := left_child.item

end

right_item: like item is

-- Value of right child

require

result := right_child.item

end

feature -- Measurement
arity: INTEGER is
  -- Number of children
  do
    if has_left then
      Result := Result + 1
    end
    if has_right then
      Result := Result + 1
    end
  ensure

child_capacity: INTEGER is 2
  -- Maximum number of children

feature -- Status report

  is_leaf, has_none: BOOLEAN is
    -- Are there no children?
    do
      Result := left_child = Void and right_child = Void
    end

  has_left: BOOLEAN is
    -- Does current node have a left child?
    do
      Result := left_child /= Void
    ensure

  has_right: BOOLEAN is
    -- Does current node have a right child?
    do
      Result := right_child /= Void
    ensure

  has_both: BOOLEAN is
    -- Does current node have two children?
    do
      Result := left_child /= Void and right_child /= Void
    ensure

feature -- Removal
6 Testing (21 Points)

6.1 General concepts (9 Points)

Put checkmarks in the checkboxes corresponding to the correct answers. Multiple correct answers are possible; there is at least one correct answer per question. A correctly set checkmark is worth 1 point, an incorrectly set checkmark is worth -1 point. If the sum of your points is negative, you will receive 0 points.

1. The purpose of performing regression testing is to check that
   a. obsolete features have been removed from recent versions of the software. [ ]
   b. changes made to the software have not introduced new bugs. [ ]
   c. bugs that were eliminated before have not re-appeared as a result of changes made to the software. [ ]

2. When doing black-box testing, the tester
a. uses the specification of the software under test. □
b. cannot use the specification, because then he would be doing white-box testing. □
c. can inspect the implementation of the features exported to ANY. □

3. If a routine $r_1$ calls another routine $r_2$ without satisfying its precondition
   a. there is a bug in $r_1$. □
   b. there is a bug in $r_2$. □
   c. there are bugs both in $r_1$ and in $r_2$. □

4. If a routine $r_1$ of a class $A$ calls a routine $r_2$ of a class $B$ (where there is no inheritance relationship between $A$ and $B$), and, when $r_2$ finishes executing, it does not fulfill the invariant of class $B$, then
   a. there is a bug in $r_1$. □
   b. there is a bug in $r_2$. □
   c. there are bugs both in $r_1$ and in $r_2$. □

5. Which of the following statements are true about the term *failure*?
   a. A failure occurs when the implementation under test produces incorrect output. □
   b. A failure occurs when the execution of the software under test takes longer than the specified time. □
   c. A failure is a problem in the source code (incorrect or missing code). □

6. Mutation testing involves
   a. changing a test case so that it exercises a different part of the software under test than the original. □
   b. introducing bugs in the software under test to see if a test suite finds them. □
   c. changing both the test suite and the software under test so that we increase test coverage. □

7. Which steps of the testing process does JUnit automate?
   a. generation of input values. □
   b. test execution. □
   c. the oracle. □

### 6.2 Contract-based testing (12 Points)

Define contract-based testing and then discuss whether it is an efficient way of finding bugs in the software. Show both its strengths and weaknesses.