Software Architecture Exam

Spring Semester 2010
Prof. Dr. Bertrand Meyer, Dr. Michela Pedroni
Date: 1 June 2010

Family name, first name: .....

Legi-Nr: .....

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: 105 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 must 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).
- Please **immediately** tell the supervisors of the exam if you feel disturbed during the exam.

Good luck!

Question	Number of possible points	Points
1	13	
2	11	
3	15	
4	20	
5	17	
6	22	

1 Multiple choice questions (13 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 0.5 points, an incorrectly set checkmark is worth -0.5 points.

Example:

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

True	False		Statement
☒	☐	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.

1.1 Which of the following statements are true and which are false?

True	False		Statement
☐	☐	a.	The *MVC* pattern completely removes coupling between the model, view, and controller.
☐	☐	b.	In the *state* pattern, you usually set the state only once.
☐	☐	c.	In the *strategy* pattern, the strategy is independent of the surrounding context.
☐	☐	d.	In the *strategy* pattern, if a strategy does not work, there is a fallback to the next strategy (if any).
☐	☐	e.	In the *chain of responsibility* pattern, if a handler does not handle an object, there is a fallback to the next handler (if any).
☐	☐	f.	The *abstract factory* pattern uses the *factory method* pattern for the creation of individual products.
☐	☐	g.	In the *flyweight* pattern, you always need to have shared and unshared flyweight objects.

1.2 Which of the following statements are true and which are false for agile methods?

True	False		Statement
☐	☐	a.	In SCRUM, the length of a Sprint may vary.
☐	☐	b.	In SCRUM, after every iteration the developers make a demo of their software.
☐	☐	c.	In SCRUM, work may be added and removed flexibly throughout a SCRUM Sprint.
☐	☐	d.	In XP (eXtreme Programming), every developer chooses his tasks to work on.
☐	☐	e.	XP requires customer representatives to be on site.
☐	☐	f.	Using pair programming, the more advanced programmer of the team always writes code while the other watches and learns from him or her.

1.3 Which of the following statements are true and which are false for concurrent computation?

True	False		Statement
☐	☐	a.	A thread can share memory with other threads.
☐	☐	b.	One can always call a feature on a **separate** attribute in SCOOP.
☐	☐	c.	The *join* method applicable to Java threads implements a synchronization mechanism.
☐	☐	d.	Every SCOOP program is free of data races.
☐	☐	e.	Deadlock is possible with Java threads and not possible in SCOOP.
☐	☐	f.	Wait by necessity in SCOOP means that a processor may continue execution even if all needed locks are not available.

1.4 Which of the following statements are true and which are false for CMMI?

True	False		Statement
☐	☐	a.	CMMI level 2 requires characterizing processes for organizations instead of individual projects.
☐	☐	b.	CMMI level 3 requires characterizing processes for organizations instead of individual projects.
☐	☐	c.	CMMI level 4 requires characterizing processes for organizations instead of individual projects.
☐	☐	d.	Organizations at the CMMI Maturity Level 1 are ready for ISO 9001:2000 registration with minor adjustments.
☐	☐	e.	Organizations at the CMMI Maturity Level 2 are ready for ISO 9001:2000 registration with minor adjustments.
☐	☐	f.	Organizations at the CMMI Maturity Level 3 are ready for ISO 9001:2000 registration with minor adjustments.
☐	☐	g.	Organizations at the CMMI Maturity Level 4 are ready for ISO 9001:2000 registration with minor adjustments.

2 Abstract Data Types (11 Points)

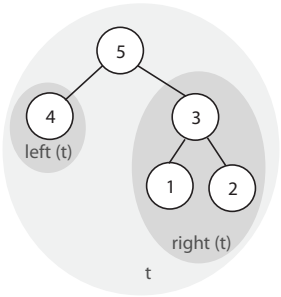
In this task you will write an abstract data type for a simple tree structure that stores integers in its nodes and whose nodes always have either no children (leaves) or two children (inner nodes). The ADT for TREE should contain the following six functions:

- *make*: Creation function that given an INTEGER argument i returns a TREE with i stored in the root node.
- *merge*: Given two arguments *t1* and *t2* of type TREE and a third argument i of type INTEGER, this function connects the two trees by adding a new root node containing i and storing *t1* as left subtree and *t2* as right subtree.
- *root*: Returns the INTEGER stored in the root node of a TREE.
- *left*: Returns the left subtree of a TREE.
- *right*: Returns the right subtree of a TREE.
- *has_children*: Returns True if the TREE has left and right subtrees, False otherwise.

Example 1

t = *make* (3)
root (t) = 3
has_children (t) = *False*
left (t) −−> not allowed
right (t) −−> not allowed

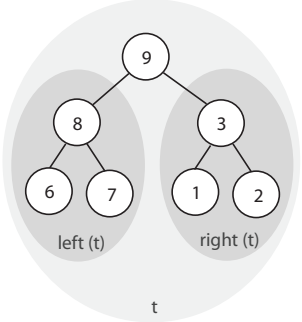
Example 2



t = *merge* (*make* (4), *merge* (*make* (1), *make* (2), 3), 5)
root (t) = 5
has_children (t) = *True*
left (t) −−> see Figure
right (t) −−> see Figure
root(*left* (t)) = 4

Example 3

t = *merge* (*merge* (*make* (6), *make* (7), 8), *right* (*merge* (*make* (4), *merge* (*make* (1), *make* (2), 3), 5)), 9)



root (t) = 9
has_children (t) = *True*
left (t) −−> see Figure
right (t) −−> see Figure
root (*left* (*right* (t))) = 1

Complete the ADT description below by filling in the missing parts in the FUNCTIONS, PRECONDITIONS, and AXIOMS sections. In the FUNCTIONS part of the ADT, you should add the appropriate function symbol in the dotted space. The axioms you propose should be sufficiently complete (but you do not need to prove sufficient completeness). The number of lines for preconditions and axioms may not correspond to the number of actual preconditions and axioms you have to provide.

TYPES TREE

FUNCTIONS

- make: INTEGER ..... TREE
- merge: TREE × TREE × INTEGER ..... TREE
- root: TREE ..... INTEGER
- left: TREE ..... TREE
- right: TREE ..... TREE
- has_children: TREE ..... BOOLEAN

PRECONDITIONS

- P1: .....
- P2: .....
- P3: .....
- P4: .....

AXIOMS

- A1: .....
- A2: .....
- A3: .....
- A4: .....
- A5: .....
- A6: .....
- A7: .....
- A8: .....

3 Design patterns and UML (15 Points)

The visitor pattern is frequently used in compilers to process expression trees. Study the following classes, which model such a scenario, and complete the UML sequence diagram for the execution of feature *make* in class *APPLICATION*. You must draw arrows for routine invocations, but not for return values.

```
indexing
  description: "An integer-valued expression tree."

deferred class
  EXPRESSION

feature

  accept (a_visitor: VISITOR)
      -- Process me with 'a_visitor'.
    deferred
    end

end
```

```
indexing
  description: "An expression tree representing an integer constant."

class
  CONSTANT

inherit
  EXPRESSION

create
  make

feature

  make (a_value: INTEGER)
      -- Set the constant value to 'a_value'.
    do
      value := a_value
    end

  value: INTEGER
    -- The constant value.

  accept (a_visitor: VISITOR)
      -- Process me with 'a_visitor'.
    do
      a_visitor.visit_constant (Current)
    end

end
```

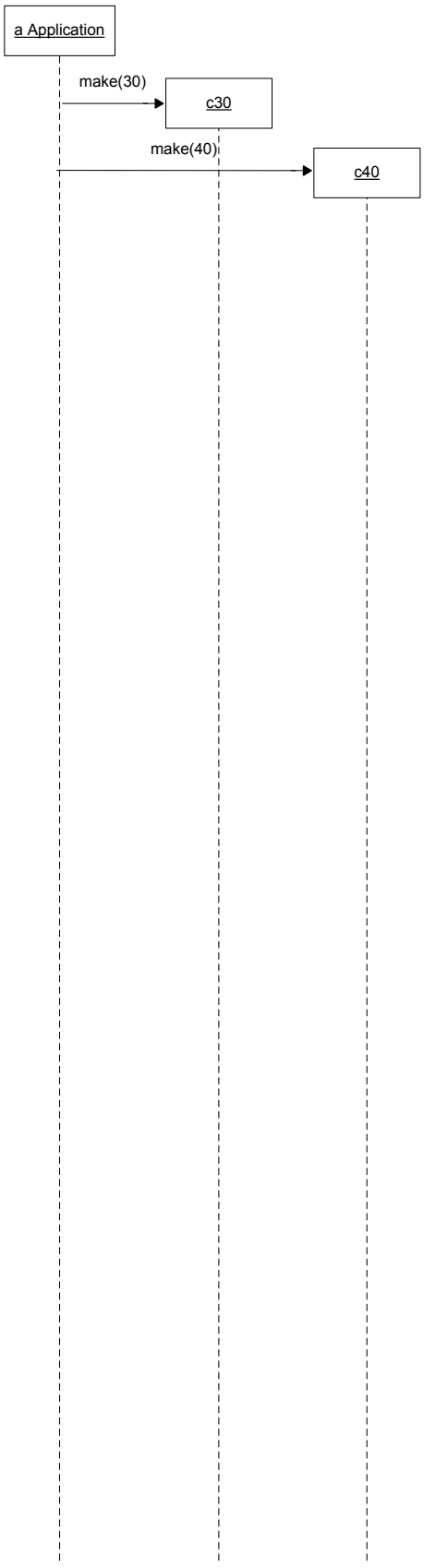
```
indexing
  description: "An expression tree representing the sum of two sub-expressions.
      "

class
  PLUS_EXPRESSION

inherit
  EXPRESSION

create
  make

feature

  make (a_left_expression, a_right_expression: EXPRESSION)
      -- Set the subexpressions to represent ' a_left_expression ' + 'a_right_expression '.
    do
      left_expression := a_left_expression
      right_expression := a_right_expression
    end

  left_expression: EXPRESSION
    -- The left operand.

  right_expression: EXPRESSION
    -- The right operand.

  accept (a_visitor: VISITOR)
      -- Process me with 'a_visitor'.
    do
      a_visitor.visit_plus_expression (Current)
    end

end
```

```
indexing
  description: "A visitor for processing expressions."

deferred class
  VISITOR

feature

  visit_constant (a_constant: CONSTANT)
      -- Process 'a_constant'.
    deferred
    end

  visit_plus_expression (a_plus_expression: PLUS_EXPRESSION)
      -- Process 'a_plus_expression'.
    deferred
    end

end
```

```
indexing
  description: "A visitor for evaluating expression trees consisting of plus nodes
      and constants."

class
  EVALUATOR

inherit
  VISITOR

create
  make

feature

  make
      -- Initialize the computed value to 0.
    do
      reset_value
    end

  reset_value
      -- Reset the computed value to 0.
    do
      value := 0
    end

  visit_constant (a_constant: CONSTANT)
      -- Add the value of 'a_constant' to the computed value.
    do
      value := value + a_constant.value
    end

  visit_plus_expression (a_plus_expression: PLUS_EXPRESSION)
      -- Add the value of 'a_plus_expression' to the computed value.
    do
      a_plus_expression.left_expression.accept (Current)
      a_plus_expression.right_expression.accept (Current)
    end

  value: INTEGER
    -- The computed value.

end
```

```
indexing
  description : "A sample use of the visitor pattern."

class
  APPLICATION

inherit
  ARGUMENTS

create
  make

feature {NONE} -- Initialization
```

```
  make
      -- Create an expression tree and compute its value.
    local
      c30, c40, c50: CONSTANT
      p1, p2: PLUS_EXPRESSION
      evaluator: EVALUATOR
      value: INTEGER
    do
      create c30.make (30)
      create c40.make (40)
      create c50.make (50)

      create p2.make (c30, c40)
      create p1.make (p2, c50)

      create evaluator.make
      p1.accept (evaluator)
      value := evaluator.value
    end

end
```


a Application
make(30)
c30
make(40)
c40

4 System Architecture (20 Points)

For the following two problems, describe the system architecture in the following form:

- Name one *architectural pattern* that you will use (not design pattern).
- Draw a diagram that describes your system architecture.
- Quickly explain in words how the system works.
- State the three most important advantages of using this architecture.
- State the two most important disadvantages of using this architecture.

4.1 E-mail Filter

An e-mail system filters incoming e-mails with a whitelist (e-mails from senders on the whitelist are accepted), a blacklist (e-mails from senders on the blacklist are deleted), and the Spamassassin tool (e-mails that do not pass this check are marked as spam). The system will run on a single-core server machine, but may be moved to a multi-core server if the load gets too high.

Architectural Pattern Name:

.....

Diagram:

.....

.....

.....

.....

.....

.....

Description:

.....

.....

.....

.....

.....

.....

Three Most Important Advantages:

.....

.....

.....

.....

.....

Two Most Important Disadvantages:

.....

.....

.....

4.2 Airplane Monitoring

In an airplane, there are many sensors: speed, altitude, cabin pressure, fuel level, etc. The monitoring system performs different checks on the sensor data. If a problem is noticed, the system either shows a warning to the pilot (e.g. low on fuel), or in a dangerous situation may react automatically (e.g. by dropping oxygen masks). The system will run on a multi-core machine and should do the checks in near real-time when new sensor data comes in.

Architectural Pattern Name:

.....

Diagram:

.....

.....

.....

.....

.....

.....

Description:

.....

.....

.....

.....

.....

.....

Three Most Important Advantages:

.....

.....

.....

.....

.....

Two Most Important Disadvantages:

.....

.....

.....

5 Testing (17 Points)

The feature *extend* of class *LINKED_LIST* is shown in the following listing, along with some of the features used in *extend*.

```
class LINKED_LIST [G]
create make

feature

  make
      -- Create an empty list.
    ensure
      count = 0
      index = 1

  first_element: like new_cell
      -- Head of list

  last_element: like first_element
      -- Tail of list

  new_cell (v: like item): LINKABLE [like item]
      -- A newly created instance of the same type as 'first_element'.

  active: like first_element
      -- Element at cursor position

  count: INTEGER
      -- Number of items in the list

  index: INTEGER
      -- Index of current cursor position (is between 1...(count+1))

  after: BOOLEAN
      -- Is there no valid cursor position to the right of cursor?
    ensure
      Result = (index = count + 1)

  is_empty: BOOLEAN
      -- Is structure empty?
    ensure
      is_empty = (count = 0)

  start
      -- Move cursor to first position.
    ensure
      index = 1

  forth
      -- Move cursor to next position.
    require
      not after
    ensure
      index = old index + 1

  put_right (v: like item)
      -- Add 'v' to the right of cursor position.
      -- Do not move cursor.
    ensure
      count = old count + 1
      index = old index
```

```
[1] extend (v: like item)
        -- Add 'v' to end.
        -- Do not move cursor.
    local
      p: like first_element
      l: like last_element
    do
[2]     p := new_cell (v)
[3]     if is_empty then
[4]       first_element := p
[5]       active := p
        else
[6]       l := last_element
[7]       if l /= Void then
[8]         l.put_right (p)
[9]         if after then
[10]          active := p
            end
          end
        end
[11]    count := count + 1
    ensure
      count = old count + 1
      index = old index
    end
```

extend adds an element to the end of a *LINKED_LIST*. *first_element* and *last_element* point to the first and the last element in the list, respectively. If the list is empty, *first_element* and *last_element* are Void. *active* points to the element at the current cursor position. If the cursor is off the list, *active* is Void.

In program analysis:

- A *definition* of a variable x (a local variable, argument or class attribute) consists of statements performing creation, initialization, assignment of a value to x or actual argument substitution if x is an argument of a feature.
- A *use* of variable x consists of statements using x without changing its value. There are two kinds of uses:
  - *P-use*: use in the predicate (decision) of an if- or loop-statement
  - *C-use*: all other uses

In the above listing, v is a passed-in argument, so line [1] is a definition of v, denoted by $v_{[1]}$, that is, the variable name followed by line number of the definition.

In the statement $p :=$ *new_cell* (v), v is C-used, so line [2] is a C-use of v, whose value is defined in line [1]. In other words, line [1] and [2] form a def-use pair for variable v. This def-use pair is denoted by $v_{[1-2]C}$, that is, the variable name, followed by two dash-separated numbers representing the definition and use location of that variable, followed by the type of use, either C or P.

Questions

(1) Please find all definitions of variables in the above listing.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

(2) Please find all def-use pairs, if any, for the definitions listed in question (1). For each def-use pair, use the described notation to indicate if it is a P-use or a C-use.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

(3) In software testing, the "all def-use criterion" is a data-flow coverage criterion. It is satisfied if all def-use pairs are examined by at least one test case. Please construct a test suite which satisfies the "all def-use criterion" for local variable *p*.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

6 Outsourcing a library system (22 Points)

The company LSN A.G. has signed a new contract with the library BB to develop a new library system. To reduce the cost of the development, the LSN A.G. has decided to outsource the implementation to India. LSN A.G. has written a first version of the requirements document. In the following we present the requirements document for the library system.

6.1 Introduction

This document describes a system for a library. The main functionalities of the system allow (1) borrowing and returning books in a library; (2) getting the list of books by a particular author or in a particular subject area, (3) finding out the list of books currently checked out by a particular borrower, (4) finding out what borrower last checked out a particular copy of a book.

6.2 Definitions, Acronyms and Abbreviations

The following table explains the terms and abbreviations used in the document:

Term/abbreviation	**Explanation**
LS	Library System
BB	Borrow Book

6.3 Glossary

Word	**Explanation**
User	Any person who uses the system and is registered within the system. It means that he or she has a user login.

6.4 Functional Requirements

6.4.1 Basics

The functional requirements are described using the following template:

ID	Defines a unique identifier of the requirement. The requirements are ordered by topic.
Title	Title of the requirement
Description	A description of the requirement.
Priority	Defines the order in which requirements should be implemented. Priorities are 1, 2, 3, and 4 (highest to lowest). Requirements of priority 1 are mandatory for the first implementation; requirements of priority 2 are mandatory for the final implementation; priority 3 is used for features that are optional but the client would like to have it; priority 4 is used for optional features.
Risk	Specifies (1) the risk of not implementing the requirement, and (2) a probability that this feature is not implemented. The first one shows how critical the requirement is to the system as a whole; the second one, the probability, is a percentage 0...100. The following risk levels are defined over the impact of not being implemented correctly. • *Critical*: it will break the main functionality of the system. The system cannot be used if this requirement is not implemented. • *High*: it will impact the main functionality of the system. Some functions of the system could be inaccessible, but the system can generally be used. • *Medium*: it will impact some system features, but not the main functionality. The system can still be used with some limitations. • *Low*: the system can be used without limitations, but with some workarounds.

6.4.2 Books

ID	R1.01
Title	Books
Description	The system stores books. The status of a book is either *available* or *borrowed*.
Priority	3
Risk	Low / 50%

ID	R1.02
Title	Books \ book copies
Description	The system also stores book copies. A book can have several copies. The status of a book copy is either *available* or *borrowed*.
Priority	3
Risk	Low / 35%

ID	R1.03
Title	Books \ book copies \ copies of a book
Description	The system must be able to show the amount of copies of a particular book. This functionality is only available to *Staff* users.
Priority	3
Risk	Low / 30%

ID	R1.04
Title	Books \ repair
Description	The system shall provide functionality to repair book copies. The status of a book copy is either *available* or *borrowed* or *broken*. If the book is in status *broken*, this functionality repairs the book by setting a string attribute to "available". This functionality is available to any user.
Priority	3
Risk	High / 10%

ID	R1.05
Title	Books \ borrow
Description	The system shall provide functionality to borrow book copies. The user selects an *available* book and borrows it. This functionality is only available to *Borrower* users.
Priority	3
Risk	Critical / 10%

ID	R1.06
Title	Books \ return
Description	The system shall provide functionality to return book copies. The user returns a book if the book status is *borrowed*. This functionality is available to any user.
Priority	3
Risk	Low / 30%

6.4.3 Users

ID	R2.01
Title	Users \ User Roles
Description	There are two kind of users: *Administrator* and *Borrower*. The *Administrator* user can access any functionality, and the *Borrower* user only the functionalities defined in R1.01, R1.02, and R1.03.
Priority	2
Risk	High / 10%

ID	R2.02
Title	Library
Description	Any *Administrator* user can check how many books are available and how many books are borrowed. Furthermore, the *Administrator* user can check how many books are broken. To borrow, or return a book, the user has to log in first.
Priority	3
Risk	Low / 40%

6.4.4 Display

ID	R3.01
Title	List of books \ By user
Description	The system shall be able to display a list of books authored or co-authored by a given author. The list shall be ordered in chronological order. If the author published more than one book in the same year, the list should be also order by the title.
Priority	1
Risk	Low / 70%

ID	R3.02
Title	List of books \ By topic
Description	The system shall be able to display a list of books in given subject area. The list shall be ordered by topic in chronological order. If there is an author who published more than one book in the same year, and area, the list should be also order by the title.
Priority	1
Risk	Low / 70%

ID	R3.03
Title	List of books \ Checked out by user
Description	The system shall be able to display a list of books that a given user has checked out. The list should be ordered by date.
Priority	1
Risk	Low / 70%

ID	R3.04
Title	List of books \ Checked out by book
Description	The system shall be able to finding out what borrower last checked out a particular copy of a book.
Priority	1
Risk	Low / 70%

6.5 Non-Function Requirements

Non-functional requirements are omitted here to keep the document short.

(This document has been signed by the client.)
15. May 2010

Task 1 (2 Points):

List the stakeholders of the system.

.....

.....

.....

.....

.....

.....

.....

.....

Task 2 (20 Points):

Given the above requirements document, find five quality goals that are not satisfied and give examples (extracted from this document). First, explain the quality goal, and then provide the example.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....