Software Architecture Exam

Summer Semester 2008
Prof. Dr. Bertrand Meyer
Date: 27 May 2008

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	8	
2	12	
3	16	
4	26	
5	16	
6	16	

1 Modularity, ADT, Design by Contract and Concurrency (8 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. **Modularity, reusablilty, ADT and design patterns.**
   a. Inheritance is a key mechanism to support the Open-Closed principle. ☐
   b. The Uniform Access principle allows a supplier to switch between storage and computation as the way to provide results to the client. ☐
   c. An ADT can be implemented as a deferred class or as an effective class. ☐
   d. Modular decomposability and modular composability imply each other. ☐
   e. It is easy to extend a composite-based design with new composite classes. ☐
   f. The visitor pattern violates the Information Hiding principle. ☐

2. **Design by Contract.**
   a. Precondition violations reveal bugs in the supplier, while postcondition violations reveal bugs in the client. ☐
   b. Class invariants can be strengthened in descendant classes. ☐
   c. During the execution of a feature, the invariant of the generating class may be violated. ☐
   d. To call a feature on an object, the client is responsible for making sure that the invariants of that object and preconditions of the feature are satisfied. ☐

3. **Concurrency with SCOOP,**

a. When assertion monitoring is turned off, an unqualified call *f(a)* with separate actual argument *a* can proceed when the object attached to *a* is reserved by the current object. ☐

b. A traitor is a separate reference attached to a non-separate object. ☐

c. Computation in a processor is sequential and will be performed in the requested order. ☐

d. Invocation of a command on a separate object is non-blocking while invocation of a query on a separate object is blocking. ☐

2 Design by Contract (12 Points)

Figure 1 shows a BON diagram of bank accounts. The class *BANK_ACCOUNT* models a bank account. This class contains a routine *withdraw* with an empty implementation. The signature of *withdraw* is *withdraw* (*v*: *INTEGER*). The precondition of this routine does not impose any restriction (any client can invoke it because its precondition is always satisfied).

The class *STUDENT_ACCOUNT* defines a student bank account. The routine *withdraw* is redefined in *STUDENT_ACCOUNT* and its precondition requires that *balance* is greater than *v*. The class *B_A_NORMAL* defines a normal bank account. It also redefines the routine *withdraw* and its precondition requires *balance* is greater than *v* plus *fee* (where *fee* is a constant).

Finally, the class *B_A_BUSINESS* defines a business bank account. This class defines an attribute *credit* storing the credit of the bank account (a positive number). The routine *withdraw* is also redefined in *B_A_BUSINESS* and its precondition requires *balance* plus *credit* is greater than *v*. In the following classes implementing this notion, complete the contracts at the locations marked by dotted lines (invariants are omitted). Furthermore, complete the redefine clauses marked by dotted lines.

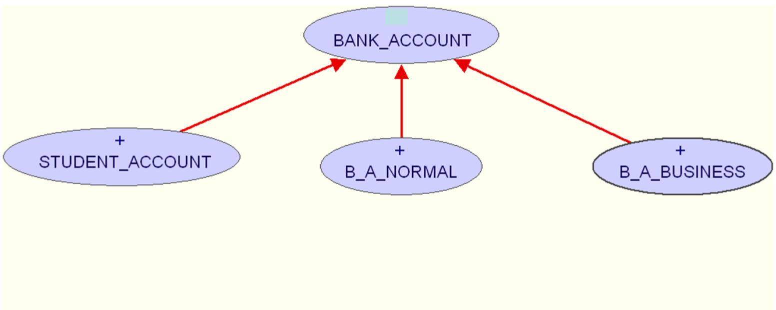

Figure 1: BON diagram of bank accounts.

```
indexing
    description: "Objects that represent a bank account."

class
    BANK_ACCOUNT

feature -- Element change
```

```
    withdraw(v: INTEGER) is
            -- withdraw v.
        require

            ..........................................................................
        do
        end


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

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

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

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

feature -- Implementation
    balance: INTEGER
end
```

```
indexing
    description: "Objects that represent a student bank account."
class
    STUDENT_ACCOUNT inherit

        BANK_ACCOUNT
            redefine

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

feature -- Element change
    withdraw(v: INTEGER) is
            -- withdraw v.
        require else

            ..........................................................................
        do
            balance := balance - v
        ensure

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

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

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

        ..............................................................................
```

```
        ..............................................................................

end
```

```
indexing
    description: "Objects that represent a normal bank account."
class
    B_A_NORMAL inherit

        BANK_ACCOUNT
            redefine

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

feature -- Element change
    withdraw(v: INTEGER) is
            -- withdraw v.
        require else

            ..........................................................................
        do
            balance := balance - v
        ensure

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

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

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

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

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

feature -- Implementation
    fee: INTEGER
end
```

```
indexing
    description: "Objects that represent a business bank account."
class
    B_A_BUSINESS inherit

        BANK_ACCOUNT
            redefine

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

feature -- Element change
    withdraw(v: INTEGER) is
            -- withdraw v.
```

```
        require else

            ......................................................................
        do
            balance := balance − v
        ensure

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


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

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

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

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

feature −− Implementation
    credit: INTEGER
end
```

3 Abstract Data Types (16 Points)

3.1 Writing an ADT for CREDIT_CARD (7 Points)

The following list describes the requirements for the implementation of a CREDIT_CARD class:

1. Every CREDIT_CARD has a limit and a debit balance.
2. The balance and the limit are recorded in "Rappen" (as INTEGERs).
3. The limit is always above 0.
4. It is always possible to retrieve the balance and the limit for any given CREDIT_CARD.
5. It is possible to settle the credit card debts (reset the debit balance to 0) and to charge the credit card with an amount (add an amount to the debit balance).
6. The balance of a CREDIT_CARD is adjusted accordingly.
7. The balance of a CREDIT_CARD should never be above the limit.
8. The amount that is charged on a credit card needs to be greater than 0.

Given is the following partial ADT description. Add type information for the functions, preconditions and axioms to complete it. Make sure to meet the requirements described above and to provide axioms that are sufficiently complete.

TYPES

CREDIT_CARD

FUNCTIONS

Creators:

- *new_card* : ....

Queries:

- *limit* : ....
- *balance* : ....

Commands:

- *settle* : ....
- *charge* : ....

PRECONDITIONS

P1 ....

P2 ....

AXIOMS

A1 ....

A2 ....

A3 ....

A4 ....

A5 ....

A6 ....

3.2 Proof of balance properties (6 Points)

Prove by structural induction of credit cards that the value returned by balance is non-negative and equal or below the value of its limit. So prove that: $balance(c) \geq 0$ and $balance(c) \leq limit(c)$ at all times.

....

....

....

....

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

3.3 Proof of sufficient completeness (3 Points)

Prove that your specification is sufficiently complete.

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

4 Design Patterns I (26 Points)

Below you find code for an imaginary car factory. The factory builds cars that consist of four wheels, a car body and an engine. The code makes use of several design patterns.

1. Identify the patterns that are used in the code fragment (12 Points)
   For each identified pattern do the following:
   - List the classes which are part of the pattern.
   - Categorize the pattern (Creational, Structural, Behavioral).
   - Give a short description of the pattern and explain what it achieves.

   . . . .

   . . . .

   . . . .

   . . . .

   . . . .

   . . . .

   . . . .

   . . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

2. Now you also want to build Mercedes cars. Extend the existing code to build Mercedes sedans, convertibles and also a Mercedes station wagon. All Mercedes cars use MERCEDES_WHEELs and MERCEDES_ENGINEs. The body is a MERCEDES_SEDAN_BODY, a MERCEDES_CONVERTIBLE_BODY or a MERCEDES_STATION_WAGON_BODY respectively. Also keep the open-closed principle in mind, i.e. do not modify existing classes. (14 Points)

Note: The classes BMW_CONVERTIBLE_WHEEL, BMW_SEDAN_WHEEL, BMW_V6_ENGINE, BMW_V8_ENGINE, BMW_CONVERTIBLE_BODY and BMW_SEDAN_BODY are direct descendants of WHEEL, ENGINE and BODY respectively with no features added. For the second assignment you can also assume that the classes MERCEDES_WHEEL, MERCEDES_ENGINE, MERCEDES_SEDAN_BODY, MERCEDES_CONVERTIBLE_BODY and MERCEDES_STATION_WAGON_BODY already exist.

```
indexing
    description: "System's root class"

class
    APPLICATION

create
    make

feature -- Initialization

    make is
            -- Run application.
        local
            l_car_factory: CAR_FACTORY
        do
            create l_car_factory.make (create {BMW_FACTORY_IMP}.make)
            l_car_factory.build_convertible
        end

end -- class APPLICATION
```

```
indexing
    description: "Abstract car factory"
```

```

class
	CAR_FACTORY

create
	make

feature {NONE} -- Initialization

	make (a_implementation: like implementation) is
			-- Create car factory with implementation 'a_implementation'
		do
			implementation := a_implementation
		end

feature -- Access

	last_car: CAR is
			-- Get the last built car
		do
			Result := implementation.last_car
		end

feature -- Operations

	build_sedan is
			-- Build a sedan (Limousine)
		do
			implementation.build_sedan
		end

	build_convertible is
			-- Build a convertible (Cabriolet)
		do
			implementation.build_convertible
		end

feature {NONE} -- Implementation

	implementation: CAR_FACTORY_IMP

end
```

```
indexing
	description: "Deferred implementation"

deferred class
	CAR_FACTORY_IMP

feature -- Access

	last_car: CAR
			-- Get the last built car

feature -- Operations

	build_sedan is
			-- Build a sedan (Limousine)
		deferred
		end

	build_convertible is
			-- Build a convertible (Cabriolet)
```

```
        deferred
        end

end
```

```
indexing
    description: "BMW factory implementation"

class
    BMW_FACTORY_IMP

inherit
    CAR_FACTORY_IMP

create
    make

feature {NONE} -- Initialization

    make is
            -- Create a BMW Factory object
        do
            create sedan_builder
            create convertible_builder
        end

feature -- Operations

    build_sedan is
            -- Build a sedan (Limousine)
        do
            sedan_builder.build
            last_car := sedan_builder.last_product
        end

    build_convertible is
            -- Build a convertible (Cabriolet)
        do
            convertible_builder.build
            last_car := convertible_builder.last_product
        end

feature -- Implementation

    sedan_builder: CAR_BUILDER[BMW_SEDAN_BODY, BMW_V8_ENGINE,
        BMW_SEDAN_WHEEL]

    convertible_builder: CAR_BUILDER[BMW_CONVERTIBLE_BODY,
        BMW_V6_ENGINE, BMW_CONVERTIBLE_WHEEL]

end
```

```
indexing
    description: "Car builder"

class
    CAR_BUILDER[G->BODY, H->ENGINE, I->WHEEL]

feature -- Access

    last_product: CAR

```

```
feature -- Build

    build is
            -- Build 'last_product'
        do
            create last_product
            build_body
            build_engine
            build_wheels
        end

feature {NONE} -- Implementation

    build_body is
            -- Build body into car
        do
            last_product . set_car_body (body_factory . new)
        end

    build_engine is
            -- Build engine into car
        do
            last_product . set_engine (engine_factory . new)
        end

    build_wheels is
            -- Build wheels into car
        do
            last_product . set_front_left_wheel (wheel_factory . new)
            last_product . set_front_right_wheel (wheel_factory . new)
            last_product . set_rear_left_wheel (wheel_factory . new)
            last_product . set_rear_right_wheel (wheel_factory . new)
        end

feature {NONE} -- Factories

    body_factory: FACTORY[G]

    engine_factory: FACTORY[H]

    wheel_factory: FACTORY[I]

end
```

```
indexing
    description: "Abstract Factory"

class
    FACTORY[G -> ANY create default_create end]

feature -- Factory methods

    new: G is
            -- Create a new object
        do
            create Result
        end

end
```

```
indexing
    description: "Objects that represent a car"
```

```
class
    CAR

feature -- Access

    front_left_wheel : WHEEL
    front_right_wheel : WHEEL
    rear_left_wheel : WHEEL
    rear_right_wheel : WHEEL

    car_body: BODY

    engine: ENGINE

feature -- Setters

    set_front_left_wheel (a_wheel: like front_left_wheel ) is
            -- Set the front left wheel
        do
            front_left_wheel := a_wheel
        end

    set_front_right_wheel (a_wheel: like front_right_wheel ) is
            -- Set the front right wheel
        do
            front_right_wheel := a_wheel
        end

    set_rear_right_wheel (a_wheel: like rear_right_wheel ) is
            -- Set the rear right wheel
        do
            rear_right_wheel := a_wheel
        end

    set_rear_left_wheel (a_wheel: like rear_left_wheel ) is
            -- Set the rear left wheel
        do
            rear_left_wheel := a_wheel
        end

    set_car_body (a_car_body: like car_body) is
            -- Set 'car_body'
        do
            car_body := a_car_body
        end

    set_engine (a_engine: like engine) is
            -- Set 'engine'
        do
            engine := a_engine
        end

end
```

```
indexing
    description: "Objects that represent a car body"

deferred class
    BODY

end
```

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

deferred class
    WHEEL

end
```

....

....

....

....

....

....

....

....

5 Web shop (16)

Assume you have written a web shop application to sell goods in Switzerland. Now your company expands to Germany and you want to use the same shop there. You have a class **SALES_ORDER** which provides the following functions:

- Allow to fill out an order
- Handle tax calculation
- Process order and print sales recipe

Unfortunately the tax calculation in Germany differs from the one in Switzerland. In this question we discuss solutions to this problem.

5.1 Copy & Paste

The first approach is to copy the code of class **SALES_ORDER** to a new class **SALES_ORDER_GERMANY** and to rewrite the the code for the the tax calculation. Do you think this is a good solution (explain why/why not)?

....

....

....

....

....

....

. . . .

. . . .

5.2 Case distinction

In this solution you define a variable **country** which returns a code for every country. Then in the tax calculation you insert a case distinction:

```
inspect country
when switzerland then
// Switzerland tax  calculation
when germany then
// Germany tax calculation
end
```

Is this in general a good solution (explain)?

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

5.3 A solution based on inheritance

Another (often used) solution would be to create two classes **SALES_ORDER_GERMANY** and **SALES_ORDER_SWITZERLAND** which inherit from **SALES_ORDER** and redefine the features used to calculate the taxes. What kind of problems could arise with this approach? (Hint: Assume there are also other differences like date format or shipping costs.)

....

....

....

....

....

....

....

....

....

....

....

....

5.4 A design pattern might help

There is a good solution based on a design pattern discussed in the lecture. Which pattern is it? Describe how you would design such a solution by naming all participants.

....

....

....

....

....

....

....

....

....

. . . .

. . . .

. . . .

6 Visitor & Composite Pattern (16 points)

6.1 Theoretical Questions (6 Points)

6.1.1 Pattern Categories (2 Point)

Which pattern-category do the following patterns belong to?

Composite pattern: . . . .

Visitor pattern: . . . .

6.1.2 Visitor and Open-Closed Principle (2 Points)

Please analyze where the visitor pattern (as introduced in the lecture) observes and/or violates the Open-Closed principle, and explain why.

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

6.2 Class Diagram (2 Points)

Draw the class-diagram of the (transparent) composite pattern, using either BON or UML notation.

6.3 Pattern Implementation (10 Points)

We now consider a composite-model of a hierarchical filesystem consisting of two types of components: *COMPOSITE_FILE* and *COMPOSITE_FOLDER*, both descendants of *COMPONENT*.

```
indexing
    description: "A generic visitable filesystem−component"

deferred class
    COMPONENT

feature −− Status report
    has_changed: BOOLEAN is deferred end
        −− Have the file−contents changed, since the last check?

    name: STRING is deferred end
        −− The name of the component

feature −− Visitor
    accept( a_visitor : VISITOR) is deferred end
        −− Accept a visitor
end
```

To perform different operations on that filesystem we want to use the visitor pattern, using the following abstract visitor.

```
indexing
    description: "Abstract visitor"

deferred class
    VISITOR

feature −− Visit
    visit_file ( a_file : COMPOSITE_FILE) is
        −− Visit a file
        require
            a_file_exists : a_file /= Void
        deferred
        end

    visit_folder ( a_folder: COMPOSITE_FOLDER) is
        −− Visit a folder
        require
            a_folder_exists : a_folder /= Void
        deferred
        end
end
```

6.3.1 Accept (3 Points)

Fill in the the code for the accept feature of *COMPOSITE_FILE* and *COMPOSITE_FOLDER*
.

```
class
    COMPOSITE_FILE
inherit
    COMPONENT
create
    make

feature {NONE} -- Initialization
    make(a_name: STRING) is
        require
            name_exists: a_name /= Void
        do
            create name.make_from_string (a_name)
        end

feature -- Status
    has_changed: BOOLEAN
    name: STRING

feature -- Visitor
    accept( a_visitor : VISITOR) is
            -- Accept a visitor
            -- TODO: Implement the accept feature, so it accepts visits
            --      from concrete visitors .
        do

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

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

```


class
    COMPOSITE_FOLDER
inherit
    COMPONENT
create
    make

feature {NONE} -- Initialization
    make(a_name: STRING) is
        require
            name_exists: a_name /= Void
        do
            create children.make
            create name.make_from_string (a_name)
        end

feature -- Status Report
    has_changed: BOOLEAN
        -- Have the contents of the folder changed since the last check?
    name: STRING

feature -- Composite
    add(a_component: COMPONENT) is
            -- Add a new child-component
        require
            a_component_exists: a_component /= Void
        do
            children.extend(a_component)
        ensure
            list_extended : children.count = old children.count + 1
        end

    remove(a_component: COMPONENT) is
            -- Remove a child component
        require
            a_component_exists: a_component /= Void
        do
            children.prune(a_component)
        end

    children: LINKED_LIST[COMPONENT]
        -- A list of all subcomponents of Current

feature -- Visitor

    accept( a_visitor : VISITOR) is
            -- Accept a visitor
            -- TODO: Implement the accept feature, so it accepts visits
            --      from concrete visitors .
        do

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

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

6.3.2 Visit (7 Points)

Complete the code for the concrete visitor *VISITOR_CHECK_CHANGE*. You have to implement a visitor that outputs the name of all *COMPOSITE_FILEs* of the filesystem that have changed since the last check.
If a file has been changed, its feature *has_changed* will be set to true by the underlying operating system.
Hint: You can use *io.put_string(a_string: STRING)* for printing a string to the console.

```
indexing
    description: "A Visitor that outputs all files that have been changed"

class
    VISITOR_CHECK_CHANGE
inherit
    VISITOR

feature -- Visit
    visit_folder (a_folder: COMPOSITE_FOLDER) is
            -- Visit a folder
            -- TODO: Implement this feature
        do

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

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

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

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

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

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

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

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

    visit_file ( a_file : COMPOSITE_FILE) is
            -- Visit a file
            -- TODO: Implement this feature
        do

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

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

```

```
            ........................................................................

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

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

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

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

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

end
```