Legi-Nr.: .......................................................

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Chair of Software Engineering
Bertrand Meyer

# Software Architecture

## *4. July 2005*

Name, First name: ..........................................................................

I confirm with my signature, that I was able to take this exam under regular conditions and that I have read and understood the instructions below.

Signature: ..........................................................................................

Instructions:

- Except for a dictionary you are not allowed to use any supplementary material.
- Use a pen (**not** a pencil)!
- Please write your legi number onto **each** sheet.
- Write your solutions directly onto the exam sheets. If you need more space for your solution ask your supervisor for more sheets. You are **not** allowed to use your own 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!

Legi-Nr.: .......................................................

| Question | Number of possible points | Points |
|---|---|---|
| 1 | 10 | |
| 2 | 18 | |
| 3 | 15 | |
| 4 | 7 | |
| 5 | 17 | |
| 6 | 20 | |
| Total | 87 | |

Grade:.........................................................................................

Legi-Nr.: ........................................................

# 1 Software Quality Principles (10 Points)

## *1.1 Correctness vs. Robustness (3 Points)*

Define Software Correctness:

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

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

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

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

Define Software Robustness:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Legi-Nr.: .......................................................

## 1.2 External quality factors (3 Points)

Software Correctness and Software Robustness are two external quality factors. List four more external quality factors (no explanation of the factors required):

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

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

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

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

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

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

## 1.3 Principles (4 Points)

Typically a DVD-Player only has a few buttons on the case and does not allow modifying the internal parameters of the electronic device. What modularity principle does this reflect? Explain the advantage of applying this principle.

Name of principle: ..........................................................................................

Explanation of advantage: ...............................................................................

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

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

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

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

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

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

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

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

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

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

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

Legi-Nr.: ........................................................

# 2 Design by Contract (18 Points)

## *2.1 True or false (8 Points)*

Consider the following statements. For each of the statements tell whether it is true or false by writing "T" for true or "F" for false.

| *Answer* | *Statement* |
|---|---|
| | Contracts of software elements are explicit declarations of the element's goal that are stored separately from the software element. |
| | Preconditions are obligations for the client but benefits for the supplier. |
| | The class invariant must be satisfied before and after object creation and before and after qualified feature calls. |
| | Preconditions only have to hold for qualified feature calls. |
| | A descendant class may strengthen but not weaken a postcondition. |
| | A postcondition violation is a bug in the client. |
| | The invariant of a class automatically includes the invariant clauses from all its parents, "or"-ed (disjunction). |
| | The execution of a rescue clause must re-establish the class invariant unless it re-triggers the exception. |

## *2.2 Class completion (10 Points)*

Consider the class *COFFEE_MACHINE* below. Complete the creation procedure *make* and provide all **preconditions**, **postconditions**, and **invariants**. Use the given lines, but note that the number of lines provided does not indicate the number of lines of code required. Make sure that your implementation achieves the following coffee machine operating modes:

- One can only refill coffee or water if the coffee container or water tank is not completely filled up.
- One can only brew a coffee or espresso if there is enough coffee and water.

Legi-Nr.: ........................................................

```
class
    COFFEE_MACHINE

create
    make

feature -- Initialization

    make is
            -- Create a new coffee machine.
        do

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

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

            .........................................................................................................................
        ensure

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

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

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

feature -- Access

    container_capacity: INTEGER is 20
            -- Maximal number of units the coffee container can contain

    tank_capacity: INTEGER is 20
            -- Maximal number of units the water tank can contain

    coffee_units: INTEGER is 2
            -- Amount of coffee units needed to brew a coffee

    espresso_units: INTEGER is 1
            -- Amount of coffee units needed to brew an espresso

    coffee_water: INTEGER is 2
            -- Amount of water units needed to brew a coffee

    espresso_water: INTEGER is 1
            -- Amount of water units needed to brew an espresso
```

Legi-Nr.: .......................................................

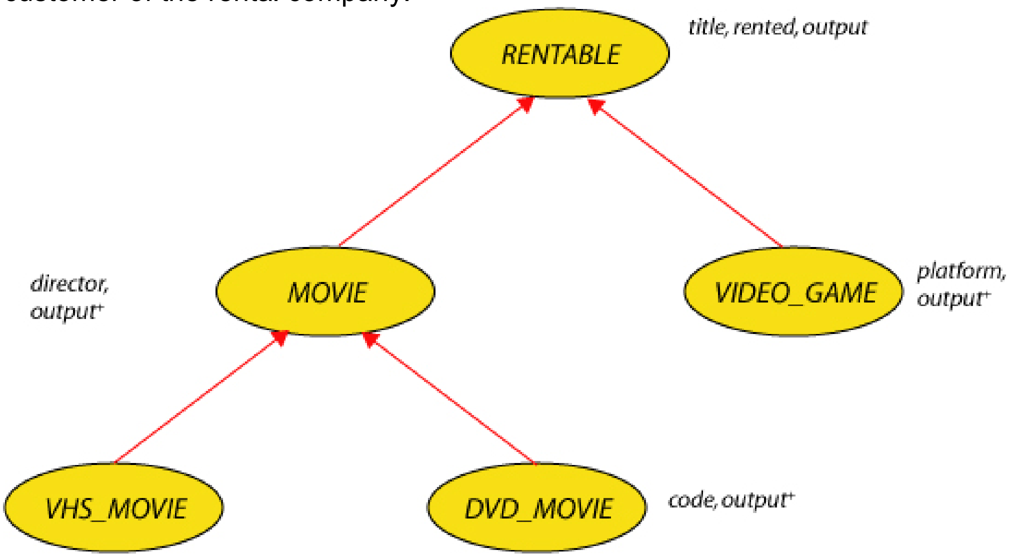
```
feature -- Measurement

    coffee_in_container: INTEGER
            -- Number of coffee units in container

    water_in_tank: INTEGER
            -- Number of water units in tank

feature -- Element change

    refill_coffee is
            -- Refill coffee container.
        require

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

            ..........................................................................................................
        do
            coffee_in_container := container_capacity
        ensure

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

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

    refill_water is
            -- Refill water tank.
        require

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

            ..........................................................................................................
        do
            water_in_tank := tank_capacity
        ensure

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

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

feature -- Basic operations

    brew_coffee is
            -- Brew coffee.
        require

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

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

```
        do
            coffee_in_container := coffee_in_container - coffee_units
            water_in_tank := water_in_tank - coffee_water
        ensure

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

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

    brew_espresso is
            -- Brew espresso.
        require

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

            .............................................................................................................
        do
            coffee_in_container := coffee_in_container - espresso_units
            water_in_tank := water_in_tank - espresso_water
        ensure

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

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

invariant

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

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

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

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

Legi-Nr.: ..........................................................

# 3 Inheritance (15 Points)

Assume we have the following inheritance hierarchy that describes the classes used for a movie and video game rental place. A *RENTABLE* is an item that can be rented by a customer of the rental company.




## 3.1 Polymorphic assignments (6 Points)

Furthermore assume that we have declared the following variables:

*video_game: VIDEO_GAME*
*dvd: DVD_MOVIE*
*rentable: RENTABLE*
*vhs: VHS_MOVIE*
*movie: MOVIE*

For each of the following assignment instructions write a "T" if and only if it is valid according to the rules of type conformance and general syntax requirements of Eiffel. If it is invalid write an "F".

| Answer | Assignment instruction |
|---|---|
| | *video_game := rentable* |
| | *dvd := movie* |
| | *rentable := movie* |
| | *rentable := dvd* |
| | *movie := vhs* |
| | *movie := video_game* |

## 3.2 Implementation (6 Points)

Below you see the class text of *RENTABLE.* Read through it carefully.

```
class RENTABLE

create
        make

feature -- Initialization

        make (an_id: like id; a_title: like title) is
                        -- Instantiate object with `a_title' and `an_id'.
                require
                        an_id_valid: an_id > 0
                        a_title_valid: a_title /= Void and then not a_title.is_empty
                do
                        id := an_id
                        title := a_title
                ensure
                        id_set: id = an_id
                        title_set: title = a_title
                end

feature -- Access

        id: INTEGER
                        -- ID number

        title: STRING
                        -- Title

        rented: BOOLEAN
                        -- Is the item rented out?

feature -- Basic operations

        output is
                        -- Output information on this item.
                do
                        io.put_string (id.out + "%N")
                        io.put_string (title + "%N")
                end

invariant

        id_valid: id > 0
        title_valid: title /= Void and then not title.is_empty

end
```

Legi-Nr.: ........................................................

Fill in the text below that is needed to override the feature *output* in the classes *MOVIE* and *DVD_MOVIE.* Read the comments of *output* to see what information should be written onto the console. Try to write the features as concisely as possible.

**class** *MOVIE* **inherit**

*RENTABLE* ..................................................................................................

**create**
*make_with_director*

**feature** -- All features

*make_with_director (an_id:* **like** *id; a_title:* **like** *title; a_director:* **like** *director)* **is**
-- Instantiate object with `a_title', `an_id' and `a_director'.
**require**
*an_id_valid: an_id > 0*
*a_title_valid: a_title /=* ***Void*** **and then not** *a_title.is_empty*
*a_director: a_director /=* ***Void*** **and then not** *a_director.is_empty*
**do**
*make (an_id, a_title)*
*director := a_director*
**ensure**
*id_set: id = an_id*
*title_set: title = a_title*
*director_set: director = a_director*
**end**

*director: STRING*
-- Director of this movie

*output* **is**
-- Output information on current movie.
-- (First output the id and a line break. Then write the title and on a
-- new line output the name of the director followed by a line
-- break.)
**do**
..............................................................................

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

..............................................................................
**end**

**invariant**

*director_valid: director /=* ***Void*** **and then not** *director.is_empty*

**end**

Legi-Nr.: ........................................................

```
class DVD_MOVIE inherit

    MOVIE   ..........................................................................................

create make_with_code

feature -- All features

    make_with_code (an_id: like id; a_title: like title;
                a_director: like director; a_code: like code) is
            -- Instantiate object with `a_title', `an_id', `a_director' and `a_code'.
        require
            an_id_valid: an_id > 0
            a_title_valid: a_title /= Void and then not a_title.is_empty
            a_director: a_director /= Void and then not a_director.is_empty
            code_valid: code >= 0 and code <= 6
        do
            make_with_director (an_id, a_title, a_director)
            code := a_code
        ensure
            id_set: id = an_id
            title_set: title = a_title
            director_set: director = a_director
            code_set: code = a_code
        end

    code: INTEGER
            -- Region code of dvd

    output is
            -- Output information on current movie.
            -- (First output the id and a line break. Then write the title and on a
            -- new line output the code followed by a line break.)
        do
            ......................................................................

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

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

invariant

    code_valid: code >= 0 and code <= 6

end
```

Legi-Nr.: .......................................................

### 3.3 Dynamic binding (3 Points)

Give the exact output (i.e. the lines that are displayed in the console window) as a result of executing the following feature *make*.

```
make is
        -- Cration procedure.
    local
        dvd: DVD_MOVIE
        vhs: VHS_MOVIE
        movie: MOVIE
    do
        create dvd.make_with_code
            (1, "House of Flying Daggers", "Yimou Zhang", 1)
        create vhs.make_with_director
            (2, "Hero", "Yimou Zhang")
        create movie.make_with_director
            (3, "Spiderman 2", "Sam Raimi")
        movie.output
        movie := vhs
        movie.output
        dvd.output
    end
```

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

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

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

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

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

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

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

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

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

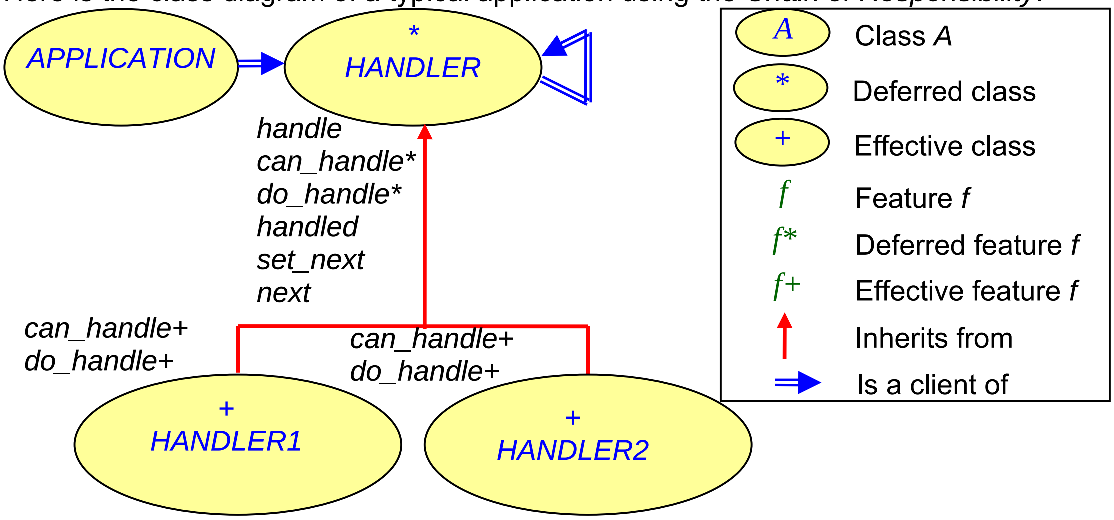
..........................................................................................................................

# 4 Adding good contracts to an existing design pattern (7 points)

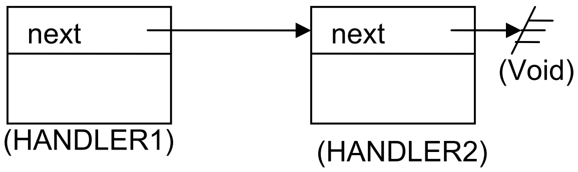
The *Chain of Responsibility* pattern addresses situations where several objects may possibly handle a client request but one does not know in advance which object will eventually treat the request.

**The Chain of Responsibility in detail:**

Here is the class diagram of a typical application using the *Chain of Responsibility*:




The *APPLICATION* sends a request to a *HANDLER*. A handler belongs to a chain of handlers (the “chain of responsibility”). For example:




If the handler receiving the request (*HANDLER1* in the previous diagram) does not know how to process this request, it simply forwards the request to its neighbor. The neighbor may be able to handle the request; if yes, it handles it, otherwise it passes the request again to the next handler on the chain. The request follows the “chain of responsibility” until one *HANDLER* is able to *handle* the request (the *HANDLER2* in the previous picture). Only one object handles the request.

A *HANDLER* only needs to know the *next* handler on the chain; it does not need to know which handler will process the request in the end. Hence less coupling between objects and more flexibility. It is also easy to change responsibilities or add or remove potential handlers from a chain because other objects do not know which handler will eventually take care of the request.

There is no guarantee that a request gets handled in the end. There may be no handler with the right qualification to handle a special request. The boolean query *handled* gives clients the ability to check whether their requests have been processed.

Legi-Nr.: ........................................................

Contracts play an important role in implementing the *Chain of Responsibility* pattern:

- They express that some objects *can_handle* requests and others cannot;
- They provide some information to clients through query *handled*.

The goal of this exercise is to equip the class *HANDLER* with the appropriate contracts.

## To do:

Add contracts (preconditions, postconditions) to the following class *HANDLER*.
Note: There is exactly one assertion clause missing per dotted line.

```
deferred class

HANDLER [G] -- G represents a request.

feature {NONE} -- Initialization

    make (a_successor: like next) is
            -- Set `next' to `a_successor'.
        do
            next := a_successor
        ensure

        ………………………………………………………………………………………
        end

feature -- Access
    next: HANDLER [G]
            -- Successor in the chain of responsibility

feature -- Status report

    can_handle (a_request: G): BOOLEAN is
            -- Can current handle `a_request'?
        deferred
        end

    handled: BOOLEAN
            -- Has request been handled?

feature -- Element change

    set_next (a_successor: like next) is
            -- Set `next' to `a_successor'.
        do
            next := a_successor
        ensure

        ………………………………………………………………………………………
        end
```

```
feature -- Basic operation

    handle (a_request: G) is
            -- Handle `a_request' if `can_handle' otherwise forward it to `next'.
            -- If `next' is void, set `handled' to False.
        do
            if can_handle (a_request) then
                do_handle (a_request)
                handled := True
            else
                    -- Cannot handle request.
                if next /= Void then
                        --  Forward it to next handler.
                    next.handle (a_request)
                    handled := next.handled
                else
                        -- Request not handled.
                    handled := False
                end
            end
        ensure

        ………………………………………………………………………………………

        ………………………………………………………………………………………

        ………………………………………………………………………………………

        end

feature {NONE} -- Implementation

    do_handle (a_request: G) is
            -- Handle `a_request'.
        require

        ……………………………………………………………………………………….
        deferred
        end

end
```

Legi-Nr.: ........................................................

# 5 Genericity (17 Points)

In this task you are asked about the differences between two list classes. Then you have to complete the interface of a class *HASH_TABLE* and a class *LIST*. Finally, you have to decide if the given Eiffel statements compile or not.

## *5.1 Two lists (4 Points)*

Consider the following two classes:

```
class LIST_1 [G]
...
feature {NONE} -- Implementation
        storage: ARRAY [G]

end
```

```
class LIST_2
...
feature {NONE} -- Implementation
        storage: ARRAY [ANY]

end
```

For each of the following statements write one of: "L1" if it is true for *LIST_1*; "L2" if it is true for *LIST_2;* "L1, L2" if it is true for both.

| Answer | Statement |
|---|---|
| | Is a generic class. |
| | Is client of a generic class. |
| | Can be used as a container for objects of types conforming to *ANY*. |
| | Can be restricted to store a list of objects of types conforming to *STRING* only. |

## 5.2 Fill in the types (8 Points)

Have a look at the following two partial class interfaces. Class *HASH_TABLE* represents data structures where arbitrary objects (*G*) can be associated with hashable objects (*H*). Class *LIST* represents a list of which no specific representation is known. In both class interfaces type names have been replaced with ........................ (a dotted line). Fill in the missing type information.

```
class interface HASH_TABLE [G, H -> HASHABLE]
feature
        has_item (v: ........................): ........................
                -- Does structure include value `v'?

        item (k: ........................): ........................
                        -- Entry of key `k'
                require
                        valid_key: valid_key (k)

        valid_key (k: ........................): ........................
                        -- Is `k' a valid key?

        put (v: ........................; k: ........................)
                        -- Associate value `v' with key `k'.
                require
                        valid_key: valid_key (k)
                ensure
                        associated: item (k) = v

    ...

end


class interface LIST [G]
feature
        has (v: ........................): ........................
                        -- Does structure include `v'?


        item: ........................
                        -- Item at current cursor position
                require
                        not_off: not off

        count: ........................
                        -- Number of items in structure

        is_empty: ........................
```

```
                    -- Is structure empty?

        off: ........................
                    -- Is there no current item?


        force (v: ........................)
                    -- Add `v' to end.
                ensure
                    new_count: count = old count + 1
                    item_inserted: has (v)

        append (s: ........................)
                    -- Append a copy of list `s'.
                require
                    argument_not_void: s /= Void
                ensure
                    new_count: count >= old count
        ...

end
```

## 5.3 Type checking (5 Points)

Given are the two classes from question 5.2 and the following local variables:

*table*: *HASH_TABLE* [*INTEGER*, *STRING*]
*list*: *LIST* [*STRING*]

(Note that class *STRING* inherits from class *HASHABLE*.)

For each of the following statements specify if it will compile or not. If it will not compile explain why not:

| | |
|---|---|
| *table*.*put* (3, "foo") | ................................................................................<br>................................................................................ |
| *table*.*put* (*list*.*item*, 3) | ................................................................................<br>................................................................................ |
| *table*.*put* (3, *list*.*item*) | ................................................................................<br>................................................................................ |
| *list*.*force* (*table*.*has* ("bar")) | ................................................................................<br>................................................................................ |
| *list*.*force* (*table*.*item* ("bar")) | ................................................................................<br>................................................................................ |

Legi-Nr.: ........................................................

# 6 Abstract Data Type (20 points)

## *6.1 Terminology (6 points):*

For each of the following statements write a "T" if it is true or write an "F" if it is false.

| Answer | Statement |
|---|---|
| | *An ADT may be defined by functions, axioms and preconditions. The axioms and preconditions express the syntax of a type.* |
| | *An ADT specification is a formal, mathematical description specifying a set of functions applicable to the instances of the type specified.* |
| | *An ADT is used to provide a basis for modularizing software with information hiding.* |
| | *Object-oriented software construction is the construction of software systems as structured collections of (possibly partial) abstract data type implementations.* |
| | *An ADT is a way of separating the specification and representation of data types. The actual implementation is not defined, and does not affect the use of the ADT.* |
| | *Total functions provide a convenient mathematical model to describe operations which are not always defined. Each operation has a precondition, stating the condition under which the operation will yield a result for any particular candidate argument.* |

Legi-Nr.: .......................................................

## 6.2 Write an ADT (8 points)

Given is the following partial interface of class *ARRAY* [*G*]. Note that contracts are omitted, but partially suggested by header comments.

**Creation procedures:**

*make* (*l*, *u*: *INTEGER*)
  -- Create a new array with the lower bound `*l*' and the upper bound `*u*'.
  -- `*l*' must be smaller or equal than `*u*'.

**Exported features:**

*lower*: *INTEGER*
  -- Lower bound

*upper*: *INTEGER*
  -- Upper bound

*item* (*i*: *INTEGER*): *G*
  -- Value at index `*i*';
  -- `*i*' must be between `*lower*' and `*upper*' (inclusive).

*put* (*v*: *G*; *i*: *INTEGER*)
  -- Replace value at index `*i*' with `*v*'.
  -- `*i*' must be between `*lower*' and `*upper*' (inclusive).

We assume that a newly created array has all its items initialized to the constant *default_value*. You may use *default_value* in your ADT without specifying it.

Write an ADT specification for this concept of array. An ADT specification for the concept of queue, as seen in the exercise, appears below and serves as illustration of the ADT notation; use the same notation to express your answer.

Legi-Nr.: ........................................................

**TYPES**

*QUEUE* [*G*]

**FUNCTIONS**

*put*: *QUEUE* [*G*] × *G* → *QUEUE* [*G*]
*remove*: *QUEUE* [*G*] –|→ *QUEUE* [*G*]
*item*: *QUEUE* [*G*] –|→ *G*
*empty*: *QUEUE* [*G*] → *BOOLEAN*
*new*: QUEUE [*G*]

**AXIOMS**

For any *x*: *G*, *q*: *QUEUE* [*G*]

*item* (*put* (*q*, *x*)) = $\begin{cases} \textit{item}\ (q)\ \textbf{if not}\ \textit{empty}\ (q) \\ x\ \textbf{if}\ \textit{empty}\ (q) \end{cases}$

*remove* (*put* (*q*, x)) = $\begin{cases} \textit{put}\ (\textit{remove}\ (q), x)\ \textbf{if not}\ \textit{empty}\ (q) \\ q\ \textbf{if}\ \textit{empty}\ (q) \end{cases}$

*empty* (*new*)
**not** *empty* (*put* (*q*, *x*))

**PRECONDITIONS**

*remove* (*q*: *QUEUE* [*G*]) **require not** *empty* (*q*)
*item* (*q*: *QUEUE* [*G*]) **require not** *empty* (*q*)

Legi-Nr.: ........................................................

TYPES:

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

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

FUNCTIONS:

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

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

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

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

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

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

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

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

PRECONDITIONS:

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

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

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

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

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

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

AXIOMS:

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

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

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

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

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

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

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

Legi-Nr.: ........................................................

### 6.3 Sufficient completeness (6 points)

Assume that someone asks you to prove that you ADT specification, as obtained in task 6.2, is “sufficiently complete”. What properties would you have to prove? (You are **not** asked in this task to do the proof, only to state what properties you would have to prove.)