Good coding practice in real life

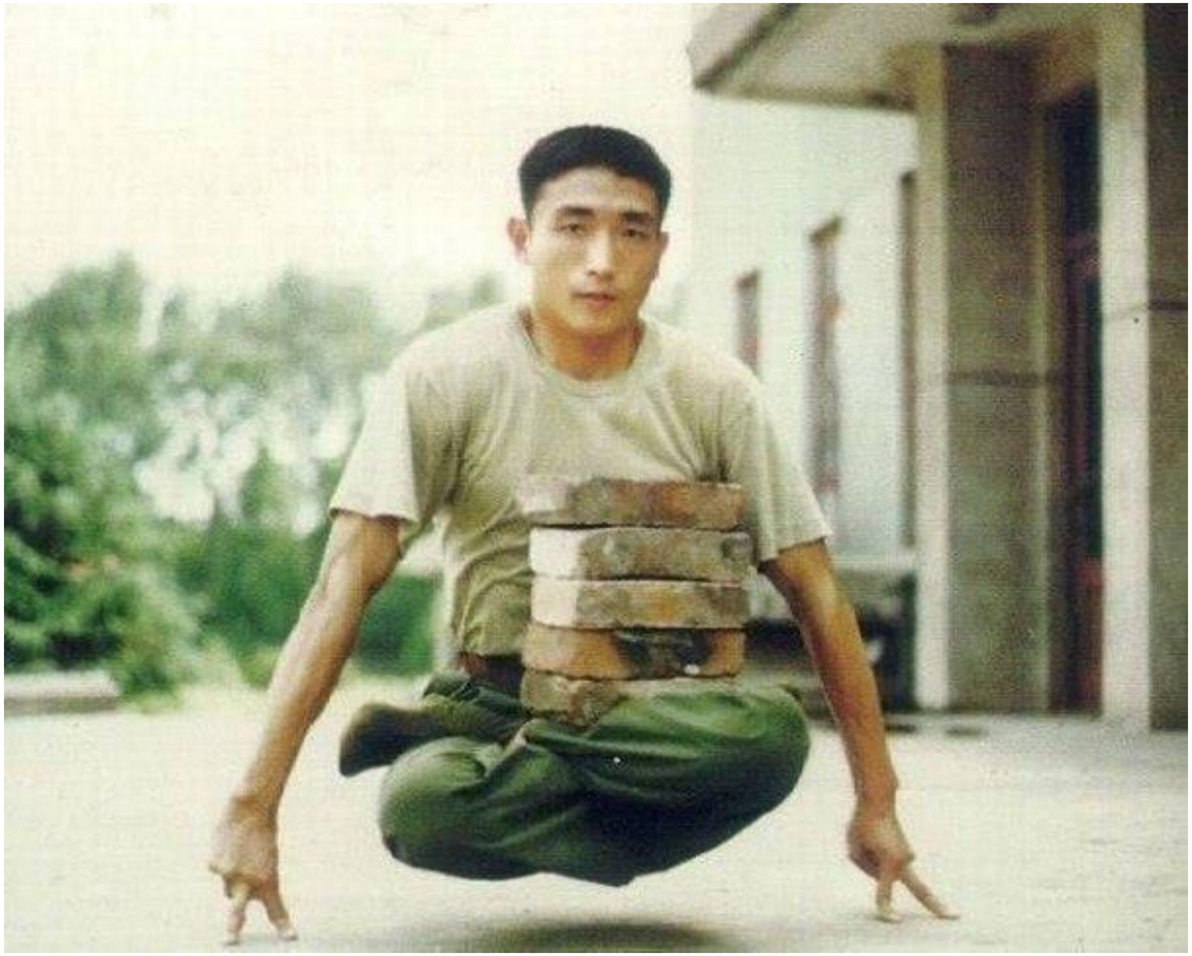
with a focus on

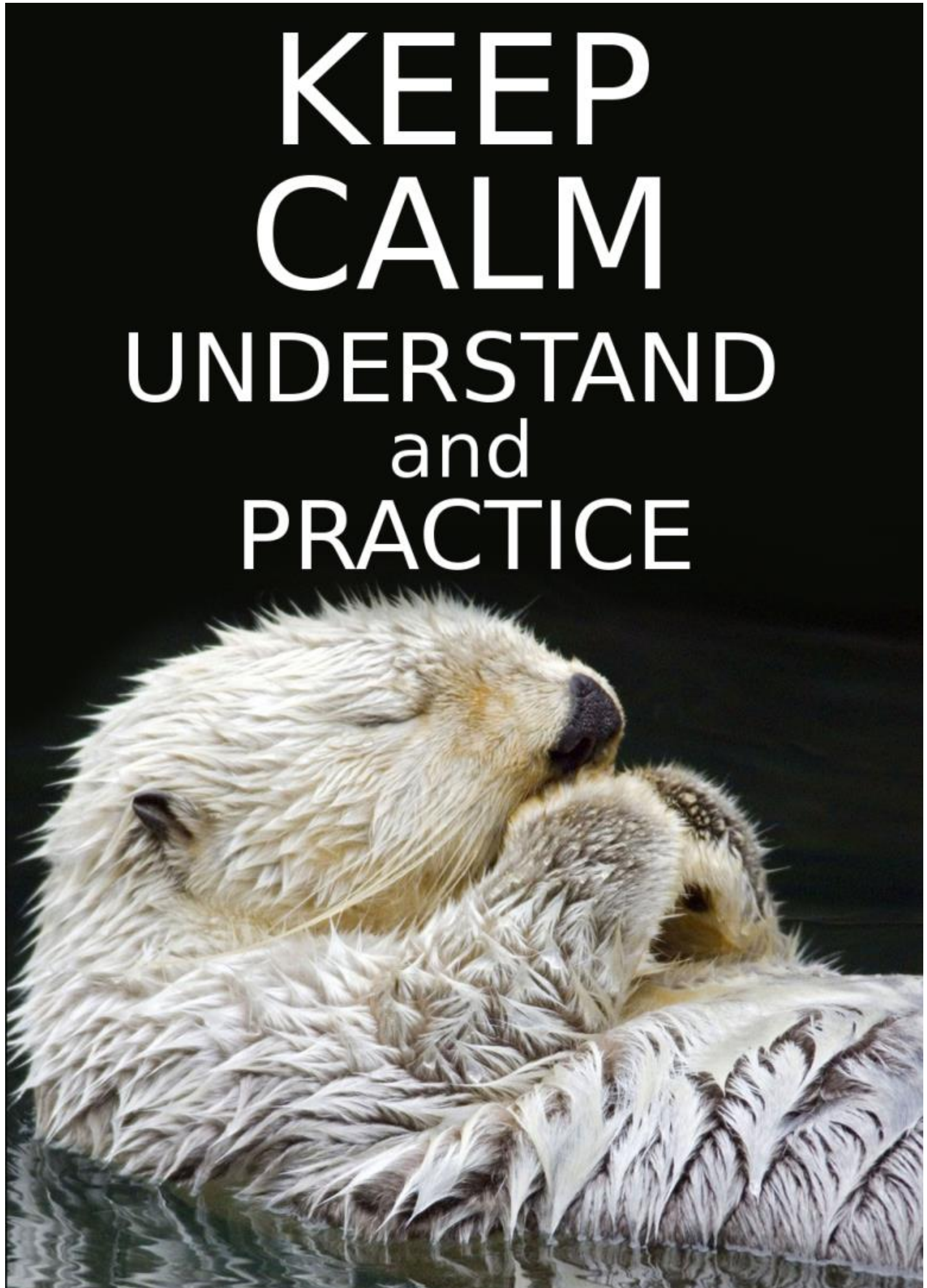
design by contract – part 2

Vladimir Alekseichenko

Good practices make life easier

Good practices are not easy




KEEP
CALM
UNDERSTAND
and
PRACTICE

Agenda

- Assertions
  - Precondition
  - Postcondition
  - Class invariant
  - **Assertion instruction**
  - **Loop invariant**
- The Eiffel language
- Monitoring assertions at run time

Assertions

- Precondition
- Postcondition
- Class invariant
- **Assertion instruction**
- **Loop invariant**

Bertrand Meyer

http://en.wikipedia.org/wiki/Bertrand_Meyer

The Eiffel language

Hello World

```
class HELLO creation make feature
    make is
    do print ("Hello World%N ") end
end
```

CNAME

```
class CNAME
    creation
        -- names of creation procedures
        -- optional
    feature
        -- declaration or definition
        -- of attributes or routines
end -- class CNAME
```

Routine

```
pname ( args ) is
    require
        -- preconditions (Boolean expressions)
    local
        -- local declarations
    do
        -- body
    ensure
        -- postconditions
    end
```

Require

```
change(balance: INTEGER): like balance is
    require
        balance: balance<=balance+78;
        n: n/=2*k;
```

Ensure

```
met is
    do
        n:=n+u;
        ensure met: old n<n
    end
```

Invariant

```
class CALC
    feature
    --....
    invariant
        upper_limit:
            hour < 24 and min < 60 and sec < 60;
        lower_limit:
            hour >= 0 and min >= 0 and sec >= 0;
end
```

Check

```
cl1 is
    do
        --...
        check
            can_be_incremented:
            clock1.sec < 59 or clock1.min < 59 or clock1.hour <23
        end
        --...
    end
```

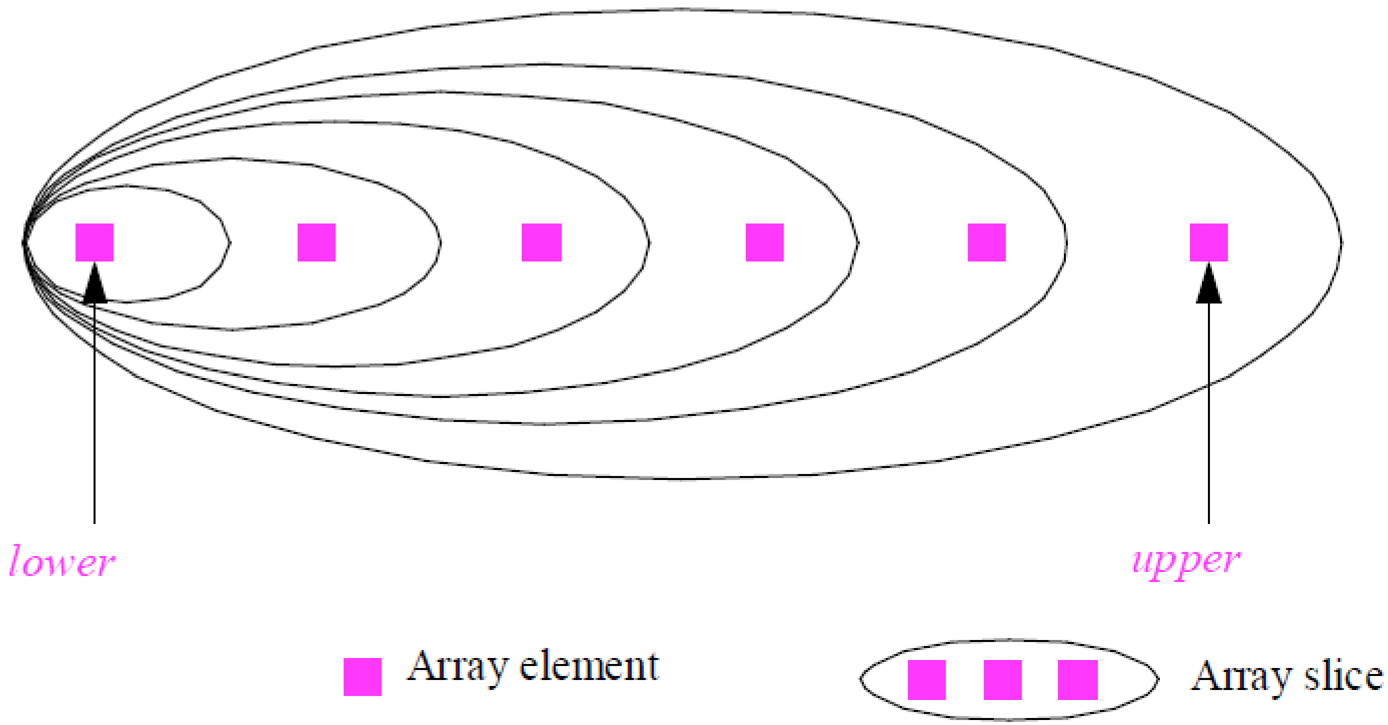
Loop syntax

```
from
    init
invariant
    inv
variant
    var
until
    exit
loop
    body
end
```

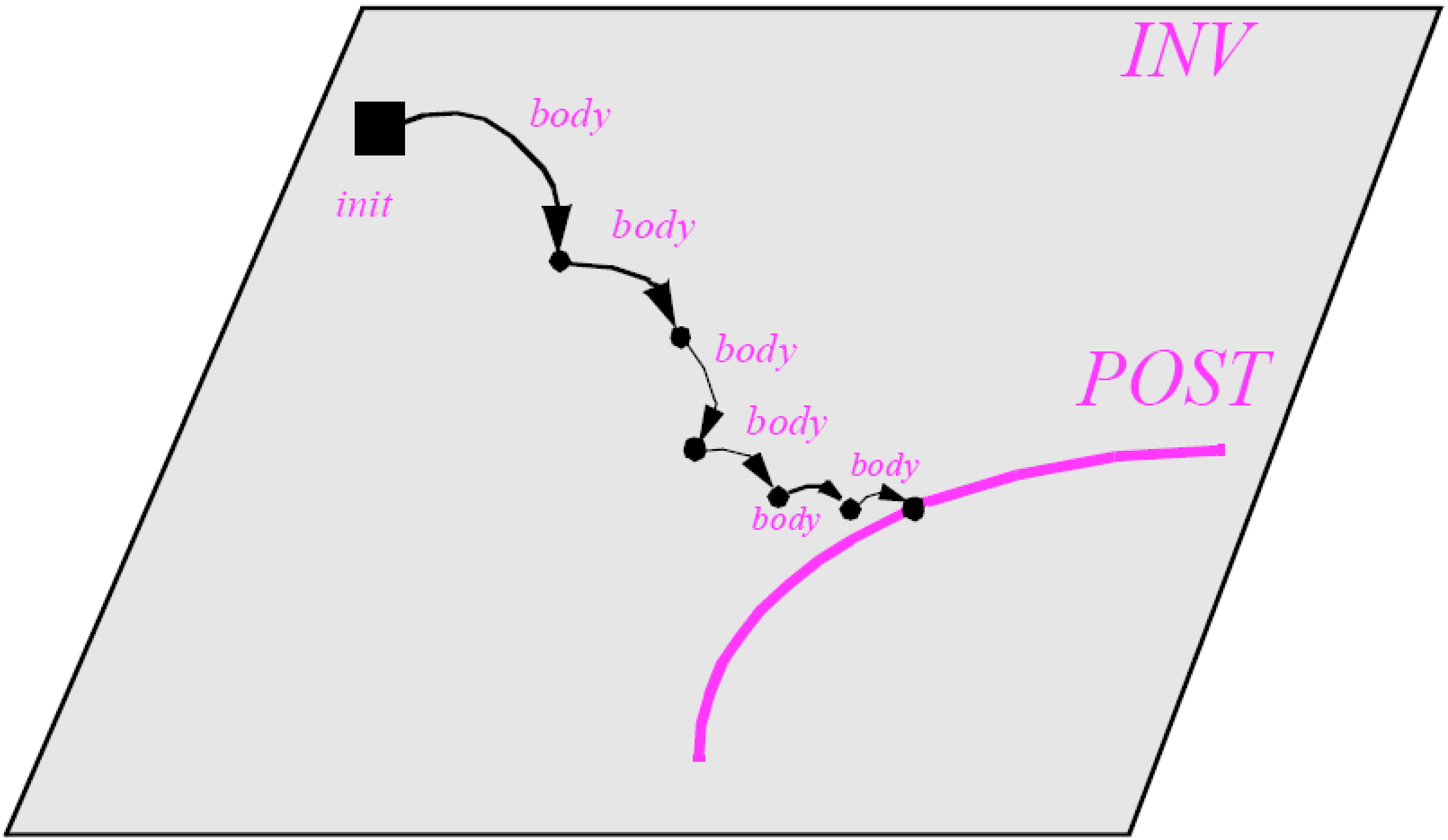
Successive approximations

```
maxarray (t: ARRAY [INTEGER]): INTEGER is
                -- The highest of the values in the entries of t
        require
                t.capacity >= 1
        local
                i: INTEGER
        do
                from
                        i := t.lower
                        Result := t @ lower
                until i = t.upper loop
                        i := i + 1
                        Result := Result.max (t @ i)
                end
end
```

Approximating an array by successive slices



A loop computation



Greatest common divisor (1)

```
gcd (a, b: INTEGER): INTEGER is
        -- Greatest common divisor of a and b
    require
        a > 0; b > 0
    local
        x, y: INTEGER
    do
        from
            x := a; y := b
        until
            x=y
        loop
            if x > y then x := x – y else y := y – x end
        end
            Result := x
        ensure
        -- Result is the greatest common divisor of a and b
    end
```

Greatest common divisor (2)

The **invariant** and **variant** clauses for gcd

```
invariant
    x > 0; y > 0
        -- The pair <x, y> has the same greatest
        -- common divisor as the pair <a, b>
variant
    x.max (y)
```

Loop invariants and variants

```
exp(n:REAL, p:INTEGER):REALis
    require non_neg_args: p >= 0 and n > 0
    local count:INTEGER
    do
        from
            Result := 1;
            count := 0;
            invariant count < p + 1;
            variant p - count
        until count = p
        loop
            Result := Result * n;
            count := count + 1;
        end --loop
    ensure
        -- returns n to power of p
    end-- exp
```

Including functions in assertions

```
full: BOOLEAN is
        -- Is stack full?
    do
        Result := (count = capacity)
    ensure
        full_definition: Result = (count = capacity)
    end
```

```
index_not_too_small: lower <= i
index_not_too_large: i <= upper
```

by a single clause of the form

```
index_in_bounds: correct_index (i)
```

The function definition

```
correct_index (i: INTEGER): BOOLEAN is
        -- Is i within the array bounds?
    do
        Result := (i >= lower) and (i <= upper)
    ensure
        definition: Result = ((i >= lower) and (i <= upper))
    end
```

The Eiffel language

- Require *(precondition)*
- Ensure *(postcondition)*
- Invariant *(class invariants)*
- Check *(assertion instruction)*
- Invariant, variant *(loop invariant)*

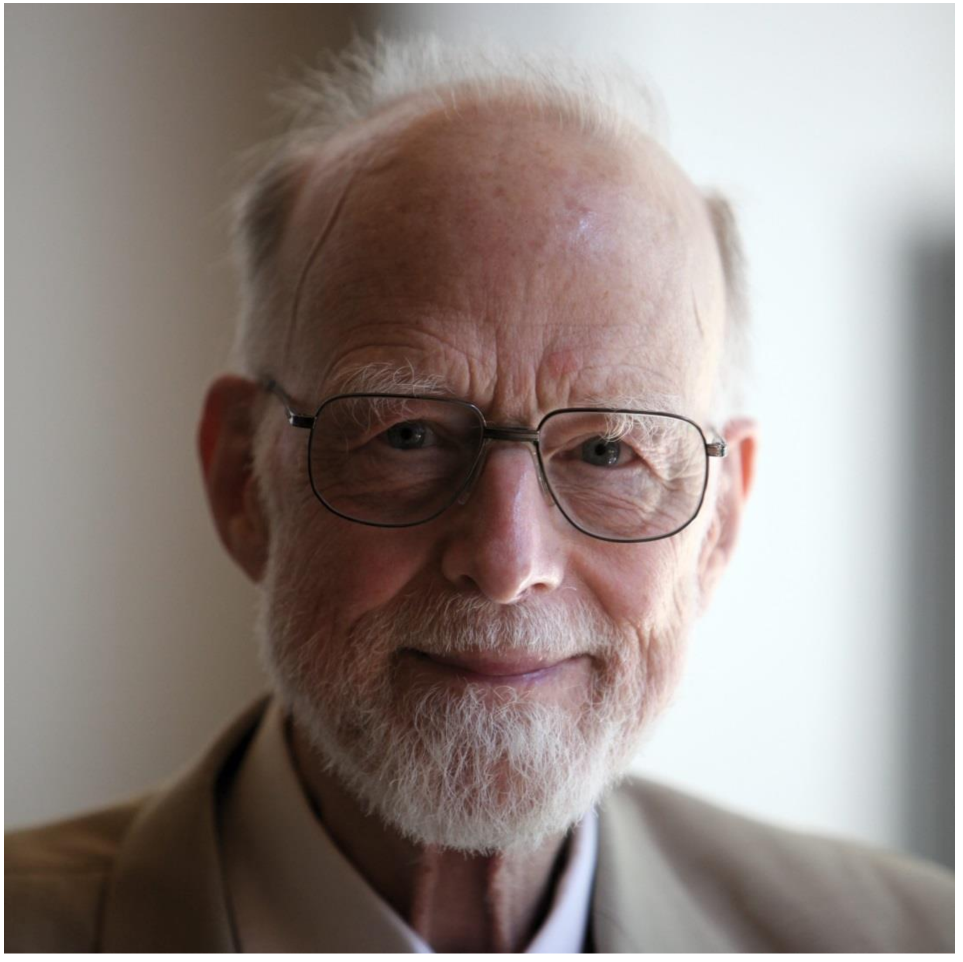
Monitoring assertions at run time

Assertion checking levels

- no
- require
- ensure
- invariant
- loop
- check (all)

How much assertion monitoring?

Charles R. Hoare



http://en.wikipedia.org/wiki/C._A._R._Hoare

„It is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous."

-- Charles R. Hoare

„What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea?"

-- Charles R. Hoare


©PrestonSpeaks.com

A correctness

- Necessary as tradeoffs between quality factors may be, one factor stands out from the rest: **correctness**.
- If the software does not perform its function, the rest is useless.

The first example

POINT (1)

```
indexing
    description: "Example Class – Creation"

class POINT -- class which supports a movable point
    creation -- designates a method to create a POINT object.
        Create

    feature
        Create (lp: LINKED_STACK [POINT]) is
                -- Create point at origin and push it onto ‘lp’
            require
                lp /= Void
            do
                lp.put (Current) -- Current is the object
            end; -- Create
```

POINT (2)

```
x, y: REAL; -- attributes of class

translate (a, b: REAL) is
        -- Move by ‘a’ horizontally, ‘b’ vertically.
    do
        x := x + a
        y := y + b
    end; -- translate

scale (factor: REAL) is
        -- Scale by ratio of 'factor'
    do
        x := factor * x
        y := factor * y
    end; -- scale
```

POINT (3)

```
        display is
                -- Output position of point
            do
                io.put_string("Current position: x=")
                io.put_real(x)
                io.put_string("; y= ")
                io.put_real(y)
                io.new_line
            end -- display

end -- class POINT
```

The second example

TIME_OF_DAY (1)

```
class TIME_OF_DAY
    -- Absolute time within a day to the nearest minute

feature
    hour: INTEGER is
            -- Hour of day, 00 to 23
        do
            Result := minutes // 60
        end -- hour

    minute: INTEGER is
            -- Minute in hour, 00 to 59
        do
            Result := minutes \\ 60
        end -- minute
```

http://www.maths.tcd.ie/~odunlain/eiffel/eiffel_course/eforb.htm

TIME_OF_DAY (2)

```
set (hh: INTEGER, mm: INTEGER) is
        -- A new time
    require
        0 <= hh and hh < 24
        0 <= mm and mm < 60
    do
        minutes := hh * 60 + mm
    ensure
        hour = hh
        minute = mm
    end -- set
```

http://www.maths.tcd.ie/~odunlain/eiffel/eiffel_course/eforb.htm

TIME_OF_DAY (3)

```
adjust (hh_by: INTEGER, mm_by: INTEGER) is
        -- Advance (+) or retard (-)
        -- either unit or both
    do
        minutes := minutes + hh_by * 60 + mm_by
        normalise
    end -- adjust
```

http://www.maths.tcd.ie/~odunlain/eiffel/eiffel_course/eforb.htm

TIME_OF_DAY (4)

```
feature {NONE}
    minutes: INTEGER
            -- Minutes since midnight

    normalise is
            -- Restore invariant after adjustment
        do
            minutes := minutes \\ 1440
            if minutes < 0 then
                minutes := minutes + 1440
            end
        end -- normalise
```

http://www.maths.tcd.ie/~odunlain/eiffel/eiffel_course/eforb.htm

TIME_OF_DAY (5)

```
invariant
    0 <= hour and hour < 24
    0 <= minute and minute < 60
    0 <= minutes and minutes < 1440

end -- class TIME_OF_DAY
```

http://www.maths.tcd.ie/~odunlain/eiffel/eiffel_course/eforb.htm

The third example

STACK (1)

```
indexing
    description: "Stacks: Dispenser structures with a Last-In, First-Out %
    %access policy, and a fixed maximum capacity"
class interface STACK [G] creation
    make

feature -- Initialization
    make (n: INTEGER) is
            -- Allocate stack for a maximum of n elements
        require
            non_negative_capacity: n >= 0
        ensure
            capacity_set: capacity = n
        end
```

STACK (2)

```
feature -- Access
    capacity: INTEGER
    -- Maximum number of stack elements

    count: INTEGER
    -- Number of stack elements

    item: G is
            -- Top element
        require
            not_empty: not empty -- i.e. count > 0
        end
```

STACK (3)

```
feature -- Status report
    empty: BOOLEAN is
            -- Is stack empty?
        ensure
            empty_definition: Result = (count = 0)
        end

    full: BOOLEAN is
            -- Is stack full?
        ensure
            full_definition: Result = (count = capacity)
        end
```

STACK (4)

```
feature -- Element change
    put (x: G) is
            -- Add x on top
        require
            not_full: not full
        ensure
            not_empty: not empty
            added_to_top: item = x
            one_more_item: count = old count + 1
        end
```

STACK (5)

```
remove is
        -- Remove top element
    require
        not_empty: not empty -- i.e. count > 0
    ensure
        not_full: not full
        one_fewer: count = old count – 1
    end
```

STACK (6)

```
invariant
    count_non_negative: 0 <= count
    count_bounded: count <= capacity
    empty_if_no_elements: empty = (count = 0)
end -- class interface STACK
```

In summary

Using assertions:

- Help in writing correct software
- Documentation aid
- Support for testing, debugging and quality assurance
- Support for software fault tolerance

Only the last two assume the ability to monitor assertions at run time.

Books

- **Object-Oriented Software Construction** by *Bertrand Meyer*
- **Design by Contract, by Example** by *Richard Mitchell, Jim McKim*