



Object-Oriented Software Construction

Bertrand Meyer

Lecture 13: Design by Contract™



Invariants

- **Invariant Inheritance rule:**
 - The invariant of a class automatically includes the invariant clauses from all its parents, "and"-ed.
- Accumulated result visible in flat and interface forms.

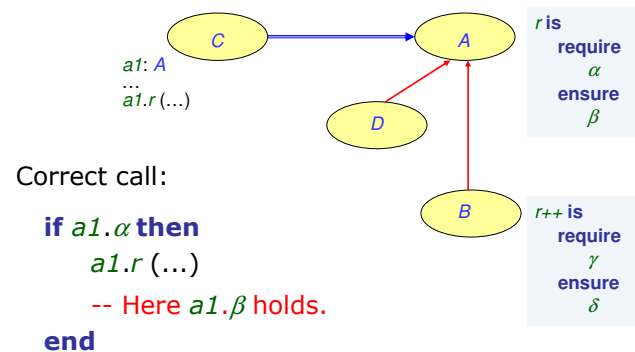


Contracts and inheritance

- **Issues: what happens, under inheritance, to**
 - Class invariants?
 - Routine preconditions and postconditions?



Contracts and inheritance



Assertion redeclaration rule

5

- When redeclaring a routine:
 - Precondition may only be kept or weakened.
 - Postcondition may only be kept or strengthened.
- Redeclaration covers both redefinition and effecting.
- Should this remain a purely methodological rule? A compiler can hardly infer e.g. that:

$$n > 1$$

implies (is stronger) than

$$n^{26} + 3 * n^{25} > 3$$

Don't call us, we'll call you

7

```
deferred class LIST [G] inherit
  CHAIN [G]
  feature
    has (x: G): BOOLEAN is
      -- Does x appear in list?
      do
        from
          start
        until
          after or else found (x)
        loop
          forth
        end
      Result := not after
    end
  end
```

Assertion redeclaration rule in Eiffel

6

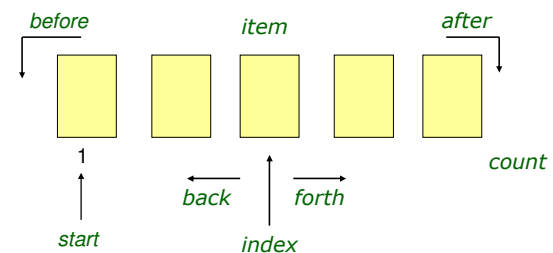
- A simple language rule does the trick!
- Redefined version may **not** have **require** or **ensure**.
- May have nothing (assertions kept by default), or

```
require else new_pre
ensure then new_post
```

- Resulting assertions are:
 - original_precondition* **or** *new_pre*
 - original_postcondition* **and** *new_post*

Sequential structures

8



Sequential structures (cont'd)

9

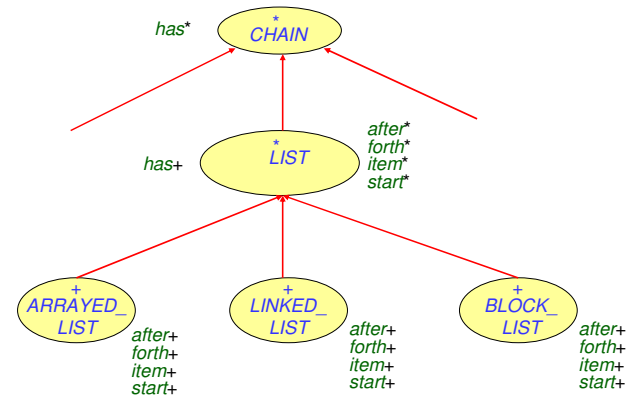
```

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

start is
  -- Move cursor to the first position.
  deferred
  ensure
    empty or else index = 1
  end
  
```

Descendant implementations

11



Sequential structures (cont'd)

10

```

index: INTEGER is
  deferred
  end

... empty, found, after, ...

invariant
  0 <= index
  index <= size + 1
  empty implies (after or before)
end
  
```

Implementation variants

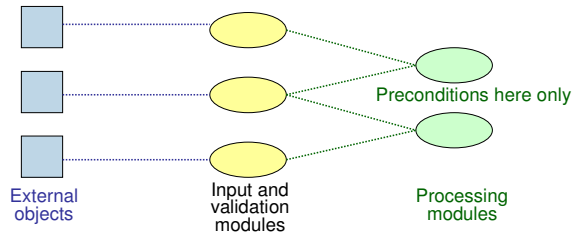
12

	start	forth	after	item (x)
Arrayed list	$i := 1$	$i := i + 1$	$i > \text{count}$	$t[i]$
Linked list	$c := \text{first_cell}$	$c := c.\text{right}$	$c := \text{Void}$	$c.\text{item}$
File	rewind	read	end_of_file	$f \uparrow$

Methodological notes

13

- Contracts are not input checking tests...
- ... but they can be used to help weed out undesirable input.
- Filter modules:



Another example

15

```

sqrt (x, epsilon: REAL): REAL is
  -- Square root of x, precision epsilon
  require
    x >= 0
    epsilon >= 0
  do
    ...
  ensure
    abs (Result ^ 2 - x) <= 2 * epsilon * Result
  end
  
```

Precondition design

14

- The client must **guarantee** the precondition before the call.
- This does not necessarily mean **testing** for the precondition.
- Scheme 1 (testing):

```

if not my_stack.is_full then
  my_stack.put (some_element)
end
  
```

- Scheme 2 (guaranteeing without testing):

```

my_stack.remove
...
my_stack.put (some_element)
  
```

The contract

16

sqrt	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Provide non-negative value and precision that is not too small.	(From postcondition:) Get square root within requested precision.
Supplier	(Satisfy postcondition:) Produce square root within requested precision.	(From precondition:) Simpler processing thanks to assumptions on value and precision.



Not defensive programming!

17

- It is **never acceptable** to have a routine of the form

```

sqrt (x, epsilon: REAL): REAL is
  -- Square root of x, precision epsilon
  require
    x >= 0
    epsilon >= 0
  do
    if x < 0 then
      ... Do something about it (?) ...
    else
      ... normal square root computation ...
    end
  ensure
    abs (Result ^ 2 - x) <= 2 * epsilon * Result
end

```



Interpreters

19

```

class BYTECODE_PROGRAM feature
  verified: BOOLEAN
  trustful_execute (program: BYTECODE) is
    require ok: verified
    do
      ...
    end
  distrustful_execute (program: BYTECODE) is
    do
      verify
        if verified then
          trustful_execute (program)
        end
      end
    end
  verify is
    do
      ...
    end
end

```



Not defensive programming

18

- For every consistency condition that is required to perform a certain operation:
 - Assign responsibility for the condition to one of the contract's two parties (supplier, client).
 - Stick to this decision: do not duplicate responsibility.
- Simplifies software and improves global reliability.



How strong should a precondition be?

20

- Two opposite styles:
 - Tolerant**: weak preconditions (including the weakest, *True*: no precondition).
 - Demanding**: strong preconditions, requiring the client to make sure all logically necessary conditions are satisfied before each call.
- Partly a matter of taste.
- But: demanding style leads to a better distribution of roles, provided the precondition is:
 - Justifiable in terms of the specification only.
 - Documented (through the short form).
 - Reasonable!

A demanding style

21

```
sqrt(x, epsilon: REAL): REAL is
  -- Square root of x, precision epsilon
  -- Same version as before
```

require

```
x >= 0
epsilon >= 0
```

do

...

ensure

```
abs(Result ^ 2 - x) <= 2 * epsilon * Result
```

end

Contrasting styles

23

```
put(x: G) is
  -- Push x on top of stack.
  require not is_full
  do
  end
```

```
tolerant_put(x: G) is
  -- Push x if possible, otherwise set impossible to
  -- True.
  do
    if not is_full then
      put(x)
    else
      impossible := True
    end
  end
```

A tolerant style

22

```
sqrt(x, epsilon: REAL): REAL is
  -- Square root of x, precision epsilon
```

require

```
True
```

do

```
if x < 0 then
```

```
  ... Do something about it (?) ...
```

```
else
```

```
  ... normal square root computation ...
```

```
  computed := True
```

```
end
```

ensure

```
computed implies
```

```
abs(Result ^ 2 - x) <= 2 * epsilon * Result
```

end



Invariants and business rules

24

- Invariants are absolute consistency conditions.
- They can serve to represent business rules if knowledge is to be built into the software.

- Form 1

invariant

```
not_under_minimum: balance >= Minimum_balance
```

- Form 2

invariant

```
not_under_minimum_if_normal:
  normal_state implies
  (balance >= Minimum_balance)
```

Loop trouble

25

- Loops are needed, powerful
- But **very hard to get right**:
 - "off-by-one"
 - Infinite loops
 - Improper handling of borderline cases
- For example: binary search feature

Computing the max of an array

27

- Approach by successive slices:

```
max_of_array (t: ARRAY [INTEGER]): INTEGER is
  -- Maximum value of array t
  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
```

The answer: assertions

26

- Use of loop variants and invariants.
- A loop is a way to compute a certain result by **successive approximations**.
- (e.g. computing the maximum value of an array of integers)

Loop variants and invariants

28

- Syntax:

```
from
  init
invariant
  inv -- Correctness property
variant
  var -- Ensure loop termination.
until
  exit
loop
  body
end
```

Maximum of an array (cont'd)

29

```
max_of_array (t: ARRAY [INTEGER]): INTEGER is
  -- Maximum value of array t
  local
    i: INTEGER
  do
    from
      i := t.lower
      Result := t @ lower
    invariant
      -- Result is the max of the elements of t at indices
      -- t.lower to i
    variant
      t.lower - i
    until
      i = t.upper
    loop
      i := i + 1
      Result := Result.max (t @ i)
    end
  end
```

Another one...

31

- **Check instruction**: ensure that a property is True at a certain point of the routine execution.
- E.g. Tolerant style example: Adding a check clause for readability.

A powerful assertion language

30

- Assertion language:
 - Not first-order predicate calculus
 - But powerful through:
 - Function calls
 - Even allows to express:
 - Loop properties

Precondition design

32

- Scheme 2 (guaranteeing without testing):

```
my_stack.remove
check
  my_stack_not_full: not my_stack.is_full
end
my_stack.put (some_element)
```



End of lecture 13