Object-Oriented Software Construction
Bertrand Meyer

Lecture 13:
Design by Contract™

Contracts and inheritance

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

Contracts and inheritance

- 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.

Correct call:

if a1.α then
  a1.r (...)
  -- Here a1.β holds.
end
**Assertion redeclaration rule**

- 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
  
  \[ n^{26} + 3 \times n^{25} > 3 \]

**Don’t call us, we’ll call you**

```plaintext
defered 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
```

**Assertion redeclaration rule in Eiffel**

- A simple language rule does the trick!
  - Redefined version may not have require or ensure.

- May have nothing (assertions kept by default), or
  
  ```plaintext
  require else new_pre
  ensure then new_post
  ```

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

**Sequential structures**

```plaintext
before item after
1 back forth index
```

- `start`
Sequential structures (cont’d)

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

index: INTEGER is
   deferred
end
... empty, found, after, ...

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

Descendant implementations

Implementation variants

<table>
<thead>
<tr>
<th></th>
<th>start</th>
<th>forth</th>
<th>after</th>
<th>item (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrayed list</td>
<td>i := 1</td>
<td>i := i + 1</td>
<td>i &gt; count</td>
<td>i[i]</td>
</tr>
<tr>
<td>Linked list</td>
<td>c := first_cell</td>
<td>c := c.right</td>
<td>c := Void</td>
<td>c.item</td>
</tr>
<tr>
<td>File</td>
<td>rewind</td>
<td>read</td>
<td>end_of_file</td>
<td>1↑</td>
</tr>
</tbody>
</table>

start forth after item (x)
Methodological notes

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


Another example

\[ \text{sqrt} \ (x, \ \text{epsilon}: \text{REAL}) : \text{REAL} \text{ is} \]
\[ \text{require} \]
\[ x \geq 0 \]
\[ \text{epsilon} \geq 0 \]
\[ \text{do} \]
\[ \ldots \]
\[ \text{ensure} \]
\[ |\text{Result}^2 - x| \leq 2 \times \text{epsilon} \times \text{Result} \]
\[ \text{end} \]

Precondition design

- The client must guarantee the precondition before the call.
- This does not necessarily mean testing for the precondition.
- Scheme 1 (testing):
  \[ \text{if not my_stack.is_full then} \]
  \[ \text{my_stack.put (some_element)} \]
  \[ \text{end} \]
  Scheme 2 (guaranteeing without testing):
  \[ \text{my_stack.remove} \]
  ... 
  \[ \text{my_stack.put (some_element)} \]

The contract

<table>
<thead>
<tr>
<th>sqrt</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>(Satisfy precondition:) Provide non-negative value and precision that is not too small.</td>
<td>(From postcondition:) Get square root within requested precision.</td>
</tr>
<tr>
<td>Supplier</td>
<td>(Satisfy postcondition:) Produce square root within requested precision.</td>
<td>(From precondition:) Simpler processing thanks to assumptions on value and precision.</td>
</tr>
</tbody>
</table>
Not defensive programming!

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

\[
\text{sqrt}(x, \text{epsilon: REAL}) \text{ REAL is} \\
\text{require} \\
x \geq 0 \\
\text{epsilon} \geq 0 \\
\text{do} \\
\text{if} x < 0 \text{ then} \\
\ldots \text{ Do something about it (?)} \ldots \\
\text{else} \\
\ldots \text{ normal square root computation} \ldots \\
\text{end} \\
\text{ensure} \\
\text{abs} (\text{Result} + 2 - x) \leq 2 \times \text{epsilon} \times \text{Result} \\
\text{end}
\]

Interpreters

\[
\text{class BYTECODE_PROGRAM feature} \\
\text{verified: BOOLEAN} \\
\text{trustful\_execute (program: BYTECODE) is} \\
\text{require} \\
\text{ok: verified} \\
\text{do} \\
\text{end} \\
\text{distrustful\_execute (program: BYTECODE) is} \\
\text{do} \\
\text{verify} \\
\text{if verified then} \\
\text{trustful\_execute (program)} \\
\text{end} \\
\text{end} \\
\text{verify is} \\
\text{do} \\
\text{end} \\
\text{end}
\]

Not defensive programming

- 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?

- Two opposite styles:
  - Tolerant: weak preconditions (including the weakest, \textit{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

```plaintext
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

```plaintext
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

```plaintext
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

- Invariants are absolute consistency conditions.
- They can serve to represent business rules if knowledge is to be built into the software.
- Form 1
  ```plaintext
  invariant
      not_under_minimum: balance >= Minimum_balance
  ```
- Form 2
  ```plaintext
  invariant
      not_under_minimum_if_normal:
          normal_state implies
              (balance >= Minimum_balance)
  ```
**Loop trouble**

- 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**

- Approach by successive slices:
  ```plaintext
  max_of_array (t: ARRAY [INTEGER]): INTEGER is
  local
    i: INTEGER
  do
    from
    i := t.lower
    until
    i = t.upper
    loop
      i := i + 1
      Result := Result.max (t @ i)
    end
  end
  ```

**The answer: assertions**

- 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**

- Syntax:
  ```plaintext
  from
  init
  invariant
  inv
  variant
  var
  until
  exit
  loop
  body
  end
  ```
  - Correctness property
  - Ensure loop termination.
Maximum of an array (cont’d)

```pascal
max_of_array (t: ARRAY [INTEGER]): INTEGER is
  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
```

Another one...

- **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

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

Precondition design

- **Scheme 2 (guaranteeing without testing)**:
  ```pascal
  my_stack.remove
  check
    my_stack_not_full: not my_stack.is_full
  end
  my_stack.put (some_element)
  ```
End of lecture 13