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?
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

Correct call:

if \( a1.\alpha \) then

\[ a1.r (...) \]

-- Here \( a1.\beta \) 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

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

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

- Resulting assertions are:
  - `original_precondition or new_pre`
  - `original_postcondition and new_post`
Don’t call us, we’ll call you

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
    end
    Result := not after
  end
Sequential structures

- before
- item
- after

1

start

back

forth

index

count
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
Sequential structures (cont’d)

\[\text{index: INTEGER is}\]
\[\text{deferred}\]
\[\text{end}\]

\[\ldots \text{empty, found, after, ...}\]

invariant

\[0 \leq \text{index}\]
\[\text{index} \leq \text{size} + 1\]
\[\text{empty implies (after or before)}\]

end
Descendant implementations

- **CHAIN**
  - has*

- **LIST**
  - has+
    - has+ (ARRAYED_LIST, LINKED_LIST, BLOCK_LIST)
      - after+ (ARRAYED_LIST, LINKED_LIST, BLOCK_LIST)
      - forth+ (ARRAYED_LIST, LINKED_LIST, BLOCK_LIST)
      - item+ (ARRAYED_LIST, LINKED_LIST, BLOCK_LIST)
      - start+ (ARRAYED_LIST, LINKED_LIST, BLOCK_LIST)

- after* (CHAIN, LIST, ARRAYED_LIST)
- forth* (CHAIN, LIST, LINKED_LIST)
- item* (CHAIN, LIST, BLOCK_LIST)
- start* (CHAIN, LIST, LINKED_LIST, BLOCK_LIST)
### 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><strong>Arrayed list</strong></td>
<td>i := 1</td>
<td>i := i + 1</td>
<td>i &gt; count</td>
<td>t [i]</td>
</tr>
<tr>
<td><strong>Linked list</strong></td>
<td>c := first_cell</td>
<td>c := c.right</td>
<td>c := Void</td>
<td>c.item</td>
</tr>
<tr>
<td><strong>File</strong></td>
<td>rewind</td>
<td>read</td>
<td>end_of_file</td>
<td>f↑</td>
</tr>
</tbody>
</table>
Methodological notes

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

  ![Diagram](image)

  - External objects
  - Input and validation modules
  - Processing modules
  - Preconditions here only
Precondition design

- 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)
  ```
Another example

\(\text{sqrt} \ (x, \epsilon: \text{REAL}): \text{REAL} \) is

\[\begin{align*}
\text{require} \\
& \ x \geq 0 \\
& \ \epsilon \geq 0 \\
\text{do} \\
& \ \cdots \\
\text{ensure} \\
& \ |\ (\text{Result}^2 - x) | \leq 2 \cdot \epsilon \cdot \text{Result} \\
\text{end}
\]\n
---

*Chair of Software Engineering*
<table>
<thead>
<tr>
<th>sqrt</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></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><strong>Supplier</strong></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

```pascal
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
```

*Chair of Software Engineering*

OOSC - Summer Semester 2005
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.
class BYTECODE_PROGRAM feature

verified: BOOLEAN

trustful_execute (program: BYTECODE) is

require

ok: verified

do

end

end

distrustful_execute (program: BYTECODE) is

do

verify

if verified then

trustful_execute (program)

end

end

verify is

do

end

end
How strong should a precondition be?

- 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

\[ \text{sqrt} \ (x, \ \text{epsilon}: \ \text{REAL}): \ \text{REAL} \ \text{is} \]
\[ \quad \text{-- Square root of } x, \text{ precision epsilon} \]
\[ \quad \text{-- Same version as before} \]

\text{require}

\[
\begin{align*}
  x & \geq 0 \\
  \text{epsilon} & \geq 0
\end{align*}
\]

\text{do}

\[
\ldots
\]

\text{ensure}

\[
\text{abs} \ (\text{Result} \ ^2 - x) \leq 2 \times \text{epsilon} \times \text{Result}
\]

\text{end}
\texttt{sqrt} \,(x, \texttt{epsilon}: \texttt{REAL}): \texttt{REAL} \textbf{is}

\begin{Verbatim}
-- Square root of \(x\), precision \texttt{epsilon}
require \texttt{True}
do if \(x < 0\) then
  ... Do something about it (?) ...
else
  ... normal square root computation ...
  \texttt{computed} := \texttt{True}
end
ensure \texttt{computed implies}
\begin{itemize}
  \item \(\text{abs} (\texttt{Result}^2 - x) \leq 2 \times \texttt{epsilon} \times \texttt{Result}\)
\end{itemize}
end
Contrasting styles

\[
\text{put} \ (x: \ G) \ \text{is}
\]
\[
\begin{align*}
\text{require} & \quad \text{not is_full} \\
\text{do} & \quad \ldots \\
\text{end} & \\
\end{align*}
\]

\[
\text{tolerant\_put} \ (x: \ G) \ \text{is}
\]
\[
\begin{align*}
\text{do} & \quad \text{if not is_full then} \\
& \quad \text{put} \ (x) \\
& \quad \text{else} \\
& \quad \text{impossible} := \text{True} \\
\text{end} & \\
\end{align*}
\]

Chair of Software Engineering

OOSC - Summer Semester 2005
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

\[
\text{invariant}
\not\text{under}\_\text{minimum} : \text{balance} \geq \text{Minimum}\_\text{balance}
\]

Form 2

\[
\text{invariant}
\not\text{under}\_\text{minimum}\_\text{if}\_\text{normal} : \\
\text{normal}\_\text{state} \text{ implies} \\
(\text{balance} \geq \text{Minimum}\_\text{balance})
\]
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
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)
Computing the max of an array

- Approach by successive slices:

```pascal
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
```
Loop variants and invariants

- Syntax:

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

\[
\text{max\_of\_array}(t: \text{ARRAY}[\text{INTEGER}]): \text{INTEGER} \text{ is}
\]

\[
\text{local}
\]

\[
i: \text{INTEGER}
\]

\[
do
\]

\[
\text{from}
\]

\[
i := t.\text{lower}
\]

\[
\text{Result} := t @ \text{lower}
\]

\[
\text{invariant}
\]

\[
\text{-- Result is the max of the elements of t at indices}
\]

\[
\text{-- t.lower to i}
\]

\[
\text{variant}
\]

\[
t.\text{lower} - i
\]

\[
\text{until}
\]

\[
i = t.\text{upper}
\]

\[
\text{loop}
\]

\[
i := i + 1
\]

\[
\text{Result} := \text{Result}.\text{max}(t @ i)
\]

\[
\text{end}
\]

\[
\text{end}
\]
A powerful assertion language

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

- 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