Object-Oriented Software Construction

Bertrand Meyer
Lecture 11:

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

\[
\text{if } a1.\alpha \text{ then } a1.r (...) \text{ end}
\]

-- Here \(a1.\beta\) holds.
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

\[ n^{26} + 3 \times n^{25} > 3 \]
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 \textit{LIST} [G] inherit
  \textit{CHAIN} [G]

feature
  \textit{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
Sequential structures

before

item

after

1

back

forth

index

count

start

1

back

forth

index

count
Sequential structures (cont’d)

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

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

Chair of Software Engineering  OOSC - Summer Semester 2004
index: INTEGER is
defered
dend

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

invariant

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

deend
Descendant implementations

- CHAIN
  - has*
  - LIST
    - has+
    - ARRAYED\_LIST
      - after+
      - forth+
      - item+
      - start+
    - LINKED\_LIST
      - after+
      - forth+
      - item+
      - start+
    - BLOCK\_LIST
      - after+
      - forth+
      - item+
      - start+
## 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>t [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>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:

  - 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

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

\text{require}

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

\text{do}

\text{...}

\text{ensure}

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

end
## The contract

<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
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

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, \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

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

-- Square root of \( x \), precision \( \text{epsilon} \)
-- Same version as before

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

\begin{align*}
\text{do} \\
& ... \\
\text{ensure} \\
& \text{abs} (\text{Result}^2 - x) \leq 2 \times \text{epsilon} \times \text{Result}
\end{align*}

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

\[ put \ (x: \ G) \ is \]
\[
    \text{-- Push } x \text{ on top of stack.}
\]
\[
    \text{require not is\_full}
\]
\[
    \text{do} 
\]
\[
    \text{....}
\]
\[
    \text{end}
\]

\[ tolerant\_put \ (x: \ G) \ is \]
\[
    \text{-- Push } x \text{ if possible, otherwise set impossible to True.}
\]
\[
    \text{do} 
\]
\[
    \text{if not is\_full then}
\]
\[
    put \ (x)
\]
\[
    \text{else}
\]
\[
    \text{impossible := True}
\]
\[
    \text{end}
\]
\[
    \text{end}
\]
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
  
  $$\text{invariant}$$
  
  \[
  \text{not\_under\_minimum}: \text{balance} \geq \text{Minimum\_balance}
  \]

- Form 2
  
  $$\text{invariant}$$
  
  \[
  \text{not\_under\_minimum\_if\_normal}: \text{normal\_state \ implies \ (balance} \geq \text{Minimum\_balance)}
  \]
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
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:

```
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:

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

```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]
    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
```

---

*Chair of Software Engineering*  
OOSC - Summer Semester 2004
A powerful assertion language

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

  \[
  \text{my\_stack}\text{.remove} \\
  \text{check} \\
  \text{my\_stack\_not\_full: not my\_stack\text{.is\_full}} \\
  \text{end} \\
  \text{my\_stack\_put (some\_element)}
  \]
End of lecture 11